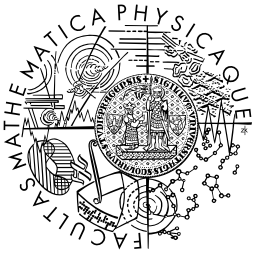


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Tomáš Pavlín

Komponování hudby pomocí programovacího jazyka

Katedra softwarového inženýrství

Vedoucí bakalářské práce:	Ladislav Maršík, M.Sc.
Studijní program:	Informatika
Studijní obor:	IPSS

Praha 2017

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Děkuji svému vedoucímu Ladislavovi Maršíkovi za odborné vedení, správné nasměrování a za mnoho užitečných informací z oboru.

Název práce: Komponování hudby pomocí programovacího jazyka

Autor: Tomáš Pavlín

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Ladislav Maršík, M.Sc., Katedra softwarového inženýrství

Abstrakt: Komponování hudby pomocí počítače přináší mnoho problémů a dá se zrealizovat mnoha různými postupy. Existující programy na skládání hudby nedávají příliš volnosti skladatelům nebo jsou příliš komplikované pro uživatele bez technického zázemí. V této práci přicházíme s intuitivním programovacím jazykem navrženým pro komponování hudby. Přikládáme také interpret tohoto jazyka, který je reprezentovaný přehledným grafickým uživatelským rozhraním umožňujícím komponovat a produkovat hudbu i uživateli bez technického či hudebního zaměření. Program přináší nový postup, kterým mohou skladatelé komponovat hudbu, umožňuje snadné vytváření hudby například do her a dá se využít k doprovodu ke zpěvu.

Klíčová slova: kompozice hudby, syntéza hudby, programovací jazyk, MIDI, formální gramatiky

Title: Music composition based on a programming language

Author: Tomáš Pavlín

Department: Department of Software Engineering

Supervisor: Ladislav Maršík, M.Sc., Department of Software Engineering

Abstract: Computer music composition brings a lot of problems which can be solved using a variety of approaches. The existing music composition programs either do not provide enough flexibility to composers or they are considerably complicated for users which do not have technical background. In this thesis, we introduce an intuitive programming language designed for music composition along with an interpreter of this language represented by user-friendly graphical interface. The interface can be utilized for music composition and production even by users without technical and musical skills. The program provides a new approach for music composition and allows an effortless music creation that can be used e.g. in game industry. In addition, the program can be used for musical accompaniment.

Keywords: music composition, music synthesis, programming language, MIDI, formal grammars

Obsah

1. Úvod

Komponování neboli skládání nové hudby má dnes mnoho motivací. Hudba se skládá nejen proto, aby se mohla hrát na koncertech, ale je klíčová jako pozadí k filmům či počítačovým hrám. Aby komponovaná hudba zněla dobře, musí odpovídat jistým pravidlům vycházejícím z hudební teorie. Jako příklad takového pravidla je zde možné uvést značnou část popových písní, které se skládají ze čtyř základních akordů, které znějí dobře, když se zahrají ve správném pořadí. Takových pravidel je ale mnohem více a dobrý hudební skladatel se vyznačuje tím, že si tato pravidla uvědomuje nebo je alespoň dokáže automaticky ve svých skladbách využívat.

Nabízí se otázka, zda je možné tato pravidla popsat tak, aby je pochopil počítač a mohl hudbu komponovat podobně jako skladatel. Kdyby se to podařilo, skladatel by mohl počítači pravidla popsat a počítač by dokázal novou hudební skladbu složit během zlomku sekundy. Navíc by pomocí stejných pravidel pak počítač různých skladeb dokázal složit velké množství a zahrát je.

1.1 Cíle práce

Ke komponování hudby a popsání pravidel využijeme programovací jazyk, který pro tyto účely navrhneme.

Tato práce usiluje o vytvoření počítačové aplikace, která by umožňovala komponování hudby pomocí programovacího jazyka a její reprodukci. Na výslednou hudbu stanovujeme dva požadavky. Za prvé chceme, aby vytvořená hudba „dobře zněla“. *Dobře znít* je pojem velmi subjektivní a tímto problémem se budeme dále v práci zabývat. Druhý požadavek je, že by jeden program naprogramovaný v hudebním programovacím jazyce měl při každém svém běhu složit pokud možno jinou skladbu.

Cílem práce je najít takové řešení, které by umožnilo v co největší míře splnit výše zmíněné požadavky a zároveň by umožnilo skladateli, coby uživateli aplikace, volnost pro své kompozice a inspirace.

Dalším cílem je zpřístupnit aplikaci i pro uživatele bez technického a hudebního zaměření a umožnit jim komponovat novou hudbu či využívat aplikaci, podobně jako se využívá hudební nástroj, k produkci hudebního podkladu ke zpěvu.

Posledním cílem je možnost aplikaci využít k automatickému vytvoření velmi dlouhých playlistů např. k účelům vývojářů počítačových her či k poslechu.

1.2 Struktura práce

V kapitole 2 si ukážeme příklady přístupů k počítačové kompozici hudby, kterými se v práci inspirujeme. Kapitola 3 obsahuje základy hudební teorie a představuje analýzu hlavních problémů, které práce řeší. V kapitole 4 je popsána základní funkcionalita uživatelského rozhraní. Kapitola 5 představuje hudební programovací jazyk, který tvoří jádro aplikace, a představuje jeho funkcionalitu

na vytvoření programu v tomto jazyce pro komponování jednoduché skladby. Kapitola 6 mluví o technologiích, které jsme v aplikaci využili a kapitola 7 seznamuje s návrhem zdrojového kódu aplikace. Nakonec v závěru shrneme práci, řekneme si, na koho je aplikace cílená a nastíníme možnosti jejích možných budoucích vylepšení.

2. Podobné práce

Algoritmické komponování hudby není ničím novým. Například již Mozart nejméně jednou použil ke komponování hudby algoritmy pomocí techniky zvané *Musikalisches Würfelspiel*, což v překladu znamená „hra v hudební kostky“ (Edwards, 2011). Tato technika používá herní kostky k náhodnému zkomponování hudby z již zkomponovaných částí.

S dnešními vědomostmi a počítači můžeme jít ještě dál. Technik k algoritmickému komponování hudby pomocí počítače dnes existuje velké množství.

2.1 Komponování hudby pomocí gramatik

Jeden z přístupů ke komponování hudby pomocí počítače je založen na formálních gramatikách. Buxton a kol. (1978) a Roads (1979) byli první, kteří techniku založenou na gramatikách aplikovali.

Ke komponování hudby a k její reprodukci přispěly vědomosti získané z podobných oborů - např. fraktálové a stochastické metody a procedurálně orientovaná a rekursivní kompozice.

2.1.1 L-systémy

Přístupů ke komponování hudby pomocí gramatik je mnoho. Přehled využívaných přísupů a prací shrnul ve své studii McCormack (1996). Zde budeme tuto metodu ilustrovat na paralelních generativních gramatikách, které jsou známé pod jménem L-systémy (Beaumont a Stepney, 2009).

Pomocí L-systémů se dají jednoduchým způsobem popisovat komplikované struktury, které mají nějakou míru opakování. Gramatiky obecně se využijí ve chvíli, kdy produkované struktury, které jsou výstupem gramatiky, něco reprezentují. Může se jednat o reprezentaci topologie, čehož se využije při geometrickém modelování, nebo, jako v našem případě, o reprezentaci not, čehož se dá využít k hudební kompozici.

Příklad

Nyní si ukážeme příklad generování gramatik podle McCormacka. Definujme bezkontextovou L-gramatiku (DOL-systém) jako uspořádanou trojici:

$$G =< \Sigma, P, \alpha >$$

kde Σ je abeceda systému, P je konečná množina přepisovacích pravidel , kde pravidlo $(a, \chi) \in P$ budeme zapisovat jako $a \to \chi$, kde $a \in \Sigma$, $\chi \in \Sigma^*$ a α je axiom, pro který platí $\alpha \in \Sigma^+$. Prázdné slovo označme ε.

Nyní začneme axiomem α a budeme ve slově paralelně přepisovat přepisovacími pravidly všechny symboly, pro které je nějaké přepisovací pravidlo definované. Ostatní necháme beze změny. Tento proces budeme několikrát opakovat a výsledné slovo pak budeme interpretovat jako informaci o melodii skladby. Melodií zde rozumíme posloupnost tónů, které je možné zahrát na nějaký hudební nástroj.

Jako příklad mějme tato přepisovací pravidla:

$$p_1 : C \to E$$
$$p_2 : E \to CGC$$
$$p_3 : G \to \varepsilon$$

Položme $\alpha = C$.

Výsledná slova po provedení pěti přepisovacích iteracích jsou uvedena v Tabulce 2.1.

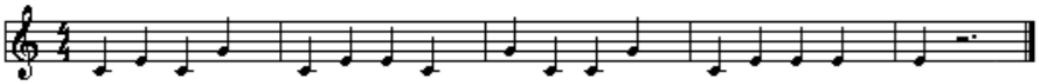
Iterace	**Slovo**
0	C
1	E
2	C G C
3	E E
4	C G C C G C
5	E E E E

Tabulka 2.1: Výsledná slova pro prvních 5 iterací gramatiky z příkladu 2.1.1.

Po zřetězení výsledných slov dostaneme:

$$CECGCEECGCCGCEEEE$$

Každý symbol slova interpretujme jako notu, čímž vznikne zkomponovaná melodie (viz Obrázek 2.1).



Obrázek 2.1: Melodie vzniklá za použití L-systému
Zdroj: McCormack (1996)

V použité gramatice jsme nepracovali s délkou not a každé vzniklé notě jsme zvolili délku noty čtvrťové. Langston (1989) ukazuje, jak lze L-systémy využít také k určení délky not a učinit tak zkomponovanou melodii komplexnější.

2.2 Live coding

Běžné programování počítačových aplikací funguje tak, že nejdříve programátor napíše zdrojový kód programu a ten pak celý zkompiluje a spustí, či interpretuje. Proces psaní kódu a spuštění programu bývá tedy časově zcela oddělený. Nicméně při komponování hudby se často využívá tzv. live coding (někdy označováno jako *on-the-fly programming* či *just in time programming*), kdy proces psaní kódu a spuštění programu funguje paralelně. Skladatel, coby programátor, tak může využít speciálního programovacího jazyka, který je zaměřen na vytváření hudby, k psaní kódu, kterým popisuje komponovanou skladbu, a ve stejný čas

tento kód interpretuje a syntetizuje hudbu. Vznikají tak vystoupení, kdy je psaní zdrojového kódu umělcem promítnuto tak, aby ho vidělo publikum, a ve stejný čas je hudba programem hrána.

V osmdesátých letech minulého století bylo live coding jako nová možnost programování pomocí interpretovaných jazyků vyzkoušeno za použití prostředí Forth a Lisp. Dnes je možné live coding praktikovat v mnoha dalších programovacích prostředích a jazycích, které byly pro tento účel nově vytvořené nebo upravené (Blackwell a Collins, 2005).

Jako příklad live coding jazyků, které slouží ke komponování či syntéze hudby, zde uvedeme známý SuperCollider (McCartney, 2002), který je odvozen od Smalltalk se syntaxí podobnou jazyku C, dnes velmi rozšířený ChucK (Wang, 2008), vizuální programovací jazyk Max[1] a nový jazyk používaný na live coding vystoupení Impromptu[2].

2.2.1 Impromptu

Impromptu je prostředí a programovací jazyk založený na Scheme, což je neprocedurální jazyk z rodiny jazyků podobných Lispu. Jazyk je navržen pro systém MacOS, kde využívá jeho zvukové komponenty AudioUnit[3]. Je používán mnohými umělci a skladateli při live coding vystoupeních.

Při vystoupení umělec ovládající programování napíše v Impromptu krátký kód programu, který posléze spustí a začne tím produkovat hudbu. Zdrojový kód pak před očima diváků upravuje a syntetizuje tak nové zvuky, přidává nové nástroje a melodie a upravuje hlasitost a tempo.

Kromě rozhraní pro audio výstup má Impromptu také vizuální komponentu, díky které může za pomoci OpenGL učinit vystoupení ještě působivější.

[1] `https://cycling74.com/products/max`

[2] `http://impromptu.moso.com.au`

[3] `https://developer.apple.com/reference/audiounit`

```
; disconnected any running graphs
(au:clear-graph)

; setup simple au graph
; piano -> output
(define piano (au:make-node "aumu" "dls " "appl"))
(au:connect-node piano 0 *au:output-node* 0)
(au:update-graph)

; hello world as a list of note pitches
; transposed down one octave
(define melody (map (lambda (c)
                      (- (char->integer c) 12))
                    (string->list "Hello World!")))

; Define a recursive function to cycle through the pitches in melod y
(define loop
   (lambda (time pitch-list)
      (cond ((null? pitch-list) (print 'done))
            (else (play-note time piano (car pitch-list) 80 10000)
                  (loop (+ time (* *second* 0.5))
                        (cdr pitch-list))))))

; start playing melody
(loop (now) melody)
```

Obrázek 2.2: Zdrojový kód programovacího jazyka Impromptu

3. Analýza problému komponování hudby

3.1 Hudební teorie

Než se budeme věnovat problému komponování hudby, vysvětlíme si několik pojmů z hudební teorie.

Za **tón** budeme považovat zvuk se stálou frekvencí. V hudbě je tón základním stavebním kamenem. Tóny budeme značit velkým písmenem, které může být následováno znakem `#` nebo `b`, které tón zvýší resp. sníží o jeden půltón. **Půltónem** značíme minimální vzdálenost výšek mezi dvěma tóny. Tónům *C,D,E,F,G,A,H* říkáme **tóny stupnice C-dur** a tuto posloupnost můžeme ekvivalentně zapsat jako *C1,D1,E1,F1,G1,A1,H1*. Uvedené sousedící tóny mají mezi sebou vzdálenost jednoho nebo dvou půltónů. Vzdálenost jednoho půltónu mezi každými sousedícími tóny má posloupnost *C,C#,D,D#,E,F,G,G#,A,A#,H*.

Oktáva je označení pro vzdálenost 12 půltónů. Pokud bychom chtěli tón zvýšit resp. snížit o jednu oktávu, zvýšili resp. snížili bychom celé číslo v jeho zápisu o 1. Pokud číslo není uvedené, je myšlené číslo 1. Tedy mezi tóny *C-1* a *C0* je vzdálenost jedné oktávy neboli 12 půltónů, mezi tóny *D0* a *D#2* vzdálenost $12 * (2 - 0) + 1 = 25$ půltónů.

Kromě výšky tónu, která je dána jeho frekvencí, má každý tón ještě délku, která určuje, jak dlouho tón zní, sílu, která je dána amplitudou, a barvu, která závisí na hudebním nástroji, který tón vydává.

Grafickému zápisu tónu se říká **nota**. Podle délky se noty dělí na noty celé, půlové, které trvají polovinu doby znění celého tónu, čtvrťové, které znějí čtvrtinu doby znění celého tónu, atd.

Pojmem **melodie** budeme označovat posloupnost tónů (např. *C,G,E,A,C2*).

Rytmus je časová složka hudby, která určuje, jakou délku mají tóny mít a kdy má tón znít či neznít. Rytmus budeme označovat schématem `<delka>[.-]+`, kde *délka* je číslo označující délku trvání tónu (noty), např. 4 pro notu čtvrťovou, 1 pro notu celou. Znak „-“ značí, že na daném místě tón o zvolené délce zazní, znak „.“ značí, že nezazní. Příkladem rytmu je `8-.--`, který je v porovnání s rytmem `16--..` dvakrát delší.

Slovy **transpozice tónu** rozumíme změnu výšky tónu o daný počet půltónů. Pro ilustraci transponováním tónu C o 5 půltónů výše vznikne tón F. **Transpozicí melodie** budeme rozumět transpozici všech tónů v melodii.

Akordem označujeme souzvuk několika tónů a budeme ho značit pomocí standardního anglického hudebního značení akordů. Tedy například akordem *B7* rozumíme souzvuk tónů *H D# F# A*.

Slovem **pedál** budeme rozumět prodlužovací pedál, který při stisku prodlužuje tón.

Pojmem **tempo** je označována rychlost hraní skladby.

Pro pochopení algoritmické kompozice hudby je klíčový pojem **hudební harmonie**. V akustice pojem znamená souzvuk a používá se ke studii současně znějících tónů a akordů. Při harmonii hrané tóny znějí hezky a příjemně. Proto je například vhodné při hraní akordu *Am* hrát nějaké tóny z tónů *A C E* a všechny od

nich transponované o násobky oktáv. Tón $A\#$ zahraný zároveň s akordem Am zní naopak disharmonicky. **Disharmonie** je opakem harmonie a označuje nehezký a nepříjemný souzvuk, který můžeme označit také slovem disonance. To ale neznamená, že je disharmonie ve všech skladbách chápaná jako něco nepatřičného. Pokud chceme, aby skladba „zněla dobře“, není nutné, aby byla harmonická, a vhodnou míru harmoničnosti skladby musí, často subjektivně, zvolit skladatel. Eldridge (2005) se problematikou harmonie zabývá detailněji.

3.2 Hudební programovací jazyk

Při řešení problému komponování hudby pomocí počítače se nabízí k tomu využít nějaký evoluční algoritmus (Tokui a kol., 2000). V této práci se ale zaměříme na takové komponování hudby, aby do procesu mohl zasahovat skladatel a využít ke komponování své inspirace. Nabízí se otázka, zda využití matematických algoritmů a teoretických pravidel nějakým způsobem autorovu inspiraci neomezí. Jak nicméně uvádí Edwards (2011), ke komponování skladatelé vždy teoretické znalosti a nacvičené postupy využívali (např. Mozart a Haydn (McCormack, 1996)) a někdy dokonce více než samotnou inspiraci. Formalizace výpočetních technik programem může osvobodit skladatelovu mysl od hudebních a kulturních klišé a vést k překvapivě originálním výsledkům.

V této práci budeme hudbu komponovat pomocí programovacího jazyka.

3.2.1 Požadavky jazyka

Pracovní návyky hudebního skladatele jsou jiné než pracovní návyky profesionálního programátora a práce umělce se odlišuje od programování ještě více (Blackwell a Collins, 2005). Protože hudební programovací jazyk cílí na skladatele a umělce, musí být co nejvíce intuitivní, snadný na pochopení a jednoduché skladby by se pomocí něho měly napsat jednoduše.

Vytvořený hudební jazyk jsme nazvali *Composing*, což je analogie k jazyku Processing[1], kterým se jazyk *Composing* inspiruje jeho jednoduchostí. Na vytvořený jazyk Composing klademe následující požadavky:

- Měl by být procedurální. Neprocedurální jazyky jsou mocné a užitečné, ale mezi lidmi nejsou tak známé a na naučení a pochopení tedy komplikovanější (viz jazyk Impromptu, 2.2.1). Procedurální jazyky jako jsou například C, Java nebo Python ovládá více lidí a naučit se s takovými vědomostmi ovládat procedurální Composing je proto snazší.

- Zdrojový kód by neměl obsahovat nic zbytného. Po jazyku chceme, aby skladatel při jeho použití psal jenom ty příkazy a instrukce, které nějakým způsobem souvisí s hudbou nebo popisují její tvoření. V jazyce proto například není žádný příkaz pro vkládání zdrojových souborů jako je `import` či `include`. Místo toho programátor jen vloží zdrojové soubory do stejné složky (případně podsložek) a ty se pak automaticky při interpretaci zřetězí. Toto automatické zřetězení zdrojových souborů je inspirováno jazykem Processing a má za cíl zjednodušit skladateli proces komponování.

[1] `https://processing.org`

- Vytvoření jednoduché skladby by se mělo docílit jednoduchým programem. Následuje ukázka celého zdrojového kódu vytvořeného jazyka, který po spuštění zahraje znělku písně *Kočka leze dírou*:

```
/* vytvoří melodii */
melody("C,D,E,F,G,G,A,A,G");

/* vytvoří rytmus, kde znak - značí
 * osminovou notu a znak . značí
 * osminovou pomlku */
rhytm("8-----.-.-.-.-");

/* namapuje tóny melodie na
 * noty rytmu a přidá do skladby */
add();
```

- Inteligentní určení typů. Přestože je vytvořený jazyk typovaný, typy proměnných a návratových hodnot metod se volí automaticky, takže programátor, coby skladatel, nemusí mít při programování jednoduchých aplikací o hodnotových typech žádné znalosti, a při komponování většiny skladeb to urychlí a zpříjemní práci.

- Od základu by měl být navržený pro komponování hudby. Kromě velkého množství standardních nadefinovaných metod, pomocí kterých se v jazyku komponuje hudba, má jazyk Composing v porovnání s ostatními programovacími jazyky zásadní rozdíl. Metoda se stejným identifikátorem může být nadefinována několikrát. Při zavolání takové metody se pak zavolá její náhodná definice. Tímto přístupem vzniká v programu velký prvek náhodnosti a běh programu je nedeterministický. To je v běžných programech nežádoucí, avšak v komponování hudby velmi užitečné. V praxi tento přístup znamená, že výsledná hudba bude nedeterministicky vytvořena a v každém běhu programu bude jiná. Tímto způsobem je možné program pouštět opakovaně a vytvářet tak různé skladby.

3.2.2 Inspirace gramatikami

Jak je zmíněno výše, mnoho postupů při komponování hudby je opřeno o formální gramatiky. Nyní bude následovat příklad, jak problém komponování hudby pomocí gramatiky převedeme do jazyka Composing.

Mějme gramatiku $G = (N, T, P, S)$, kde N je neprázdná množina neterminálních symbolů, T je konečná množina terminálních symbolů, P je konečná množina přepisovacích pravidel a $S \in N$ je počáteční symbol.

Pro příklad položme $N = \{A,B,C\}$, $T = \{c,d,e,f,g,a,h,c_2,d_2,e_2\}$ a mějme přepisovací pravidla:

$$S \to ABAC$$
$$A \to cde$$
$$A \to e_2c_2g$$
$$A \to efg$$
$$B \to ege$$
$$B \to age$$
$$C \to edc$$

Stejný problém můžeme převést do jazyka Composing:

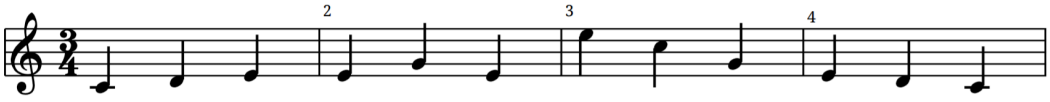
```
m = "";
A(); B(); A(); C();
print(m);

def A(){
        m += "c d e ";
}
def A(){
        m += "e2 c2 g ";
}
def A(){
        m += "e f g ";
}
def B(){
        m += "e g e ";
}
def B(){
        m += "a g e ";
}
def C(){
        m += "e d c ";
}
```

Výše uvedený zdrojový kód můžeme v jazyce napsat také kompaktnějším zápisem:

```
m = "";
A(); B(); A(); C();
print(m);

def A()
        m += "c d e ";
        m += "e2 c2 g ";
        m += "e f g ";

def B()
        m += "e g e ";
```

```
        m += "a g e ";

def C()
        m += "e d c ";
```

Uvedený kód může na výstup vypsat například `c d e e g e e2 c2 g e d c` , což odpovídá slovu jazyka popsaného gramatikou. Toto slovo můžeme interpretovat jako melodii (viz Obrázek 3.1).



Obrázek 3.1: Melodie vygenerovaná jazykem Composing při definování gramatiky

Na příkladu vidíme, že díky schopnosti jazyka definovat stejnou metodu několikrát, dosáhneme přehledným a jednoduchým způsobem výpočetní síly bezkontextové gramatiky, aniž by programátor musel být s formálními gramatikami seznámen.

Díky tomu, že můžeme v jazyce Composing používat další programátorské prostředky jako např. cykly, podmínky a proměnné, má jazyk větší výpočetní sílu (např. můžeme vygenerovat melodii, kterou budeme ve skladbě opakovat, což je v hudbě obvyklé).

3.2.3 Formát MIDI

Většina jazyků zaměřených na hudbu (viz Impromptu či ChucK, Kapitola 2) se také věnuje syntéze tónů hudby, tedy vytváření nových zvuků a nástrojů. Naskytla se otázka, zda takovou funkcionalitu přidat i do jazyka Composing. Možnost programování syntézy zvuků v jazyce Composing naimplementována není, protože jazyk klade velký důraz na jednoduchost skládání. Práce se zvukem, coby vlněním, by proces komponování učinila z hlediska uživatele jazyka složitější.

K zahraní zkomponované hudby jazyk využívá standard General MIDI (Musical Instrument Digital Interface), který reprodukovanou hudbu popisuje pomocí tónů, které jsou hrané na různé hudební nástroje. Produkovaná hudba je tímto omezena na 128 předdefinovaných hudebních nástrojů, což není mnoho, ale komponování hudby to značně zjednodušuje. Při omezení 128 hudebních nástrojů je však možné využít hudebních bank, pomocí kterých lze zvuky těchto nástrojů změnit. Ačkoli to jazyk podporuje, důraz není kladen na dynamičnost komponované skladby a zvuk zahraných tónů, ale na určení těchto tónů obecně.

General MIDI je starý standard z roku 1991, ale v dnešní době o to více rozšířený. Používá se například v elektronických klávesách.

Ačkoliv je komponování hudby Composingem inspirované hrou na klavír, je díky formátu MIDI možnost do výsledné skladby přidávat další hudební nástroje a hrát na ně současně. Do skladby je také možné přidávat bicí. Použití formátu MIDI tedy neznamená, že by výsledné skladby nemohly být komplexní. Více o formátu MIDI najdete v Kapitole 6.

3.2.4 Překlad jazyka

Nabízí se otázka, jak navržený jazyk zpracovat a spustit jeho zdrojový kód. K tomu se dá přistupovat dvěma způsoby.

Jednou možností je vytvořit kompilátor zdrojového kódu a zkompilovaný *intermediate* kód, případně strojový kód, pak spustit. Hlavní výhoda tohoto přístupu je, že je pak samotné spuštění kódu rychlejší. To se hodí zpravidla v případech, kdy napsaný kód pustíme několikrát

Druhou možností je kód načíst do paměti a interpretovat přímo v paměti bez využití *intermediate* kódu. Výhodou tohoto přístupu s ohledem na hudební jazyk je, že je pak snadnější využít jazyk k *live coding*.

Přístup, který jsme si v této práci zvolili, je druhý jmenovaný. Po zanalyzování jsme došli k závěru, že kódy programů na komponování hudby zpravidla nebudou příliš dlouhé na to, aby se vešly do paměti počítače a že problém komponování hudby není časově kritický a není nutné ho optimalizovat kompilací. Naopak z uživatelského hlediska je proces, kdy se zdrojový kód jen interpretuje, aniž by bylo nutné využívat zápisy na disk a vytvářet dočasné soubory, pochopitelnější. Ačkoli aktuální verze interpretu jazyka Composing nepodporuje live coding, je možné díky interpretaci tuto funkci v dalších verzích snadněji zavést. Více o procesu překladu jazyka je uvedeno v technické dokumentaci.

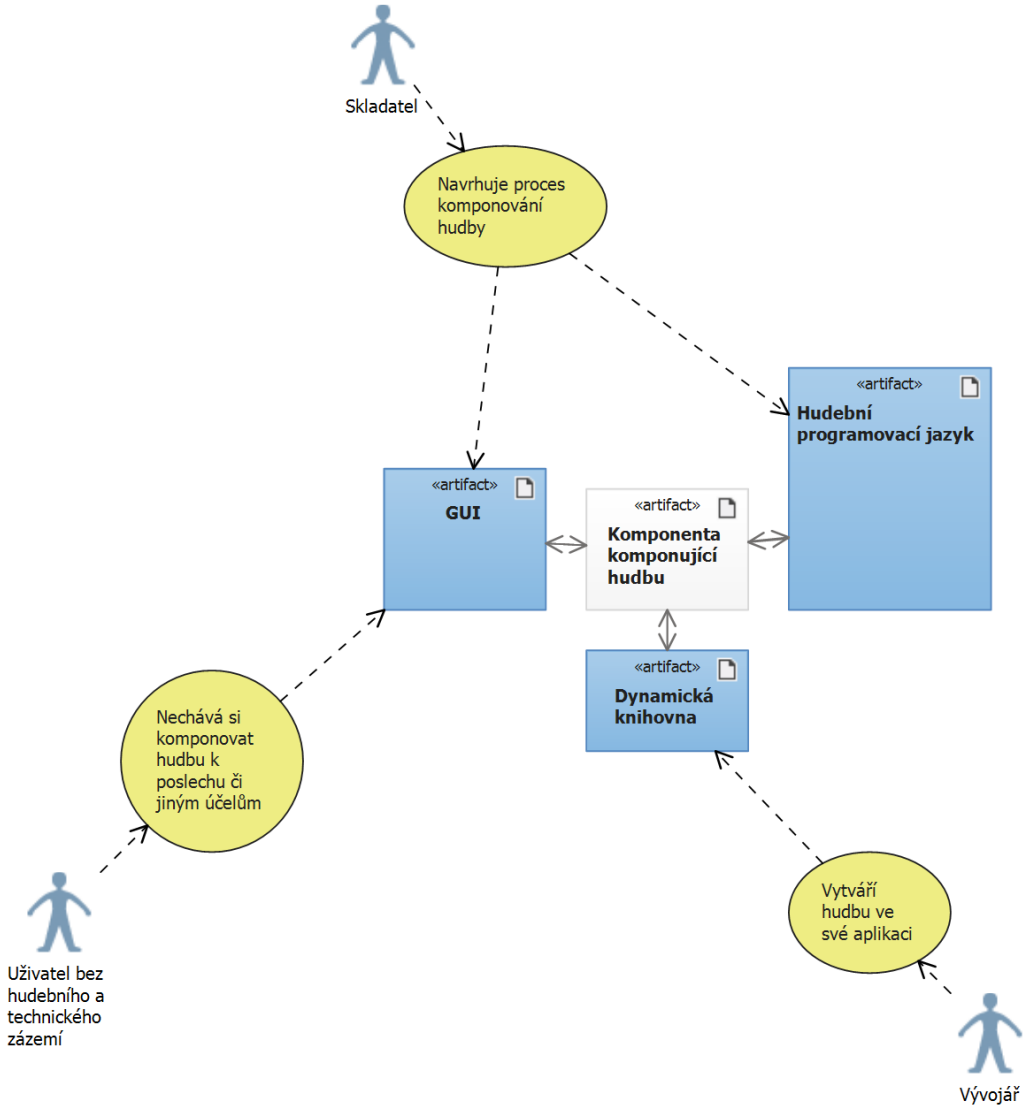
3.3 Uživatelé bez hudebního či technického zázemí

Přestože je jazyk navržen tak, aby jeho používání bylo co nejvíce intuitivní a snadné, je třeba k psaní programů, které komponují hudbu, mít hudební i programátorské dovednosti. Aplikaci ale chceme cílit nejen na hudebníky či skladatele, ale také na takové uživatele, kteří vědomosti ke skládání hudby pomocí programovacího jazyka nemají. Těmi mohou být vývojáři her či tvůrci filmů, kteří potřebují hudební materiál, a stejně tak uživatelé, kteří chtějí hudbu k poslechu či jiným osobním účelům (viz Obrázek 3.2).

Z tohoto důvodu může mít program v jazyce Processing vstupní parametry různých datových typů, kterými je možné proces komponování značně přizpůsobit. Konkrétně tak můžeme například nastavit, jaké se mají hrát akordy, které hudební nástroje se mohou použít, jak dlouhá má skladba být, jak harmonická má být a jestli má znít „smutně“ či „vesele“. Díky tomuto přizpůsobení může hudbu složit i uživatel bez hudebního či programátorského zázemí tak, že použije a nastaví již vytvořené programy jazyka, dále označované jako **šablony**.

3.3.1 Uživatelské rozhraní

Interpret jazyka je zabudovaný do grafického uživatelského rozhraní, které nese název **Song Maker**. V tomto rozhraní si i uživatel, který neumí skládat hudbu a programovat, může vybrat, který z předpřipravených šablon použije a ten pak může v závislosti na typu šablony přizpůsobit pomocí posuvníků, zaškrtávacích políček a jiných grafických elementů (viz Obrázek 4.1). Toto grafické rozhraní je celé navržené tak, že k jeho ovládání není třeba znalost jazyka Com-



Obrázek 3.2: Případy užití aplikace SongMaker a jazyka Composing

posing. Zároveň toto rozhraní také využije skladatel k interpretování a ladění svých programů.

K vytvoření grafického rozhraní jsme využili *Windows Forms*. Uživatelskou část aplikace je možné navrhnout pomocí návrhového vzoru, kterým je například model-view-controller (MVC), model-view-presenter (MVP) nebo model-view-viewmodel (MVVM).

S ohledem na technologii *Windows Forms* není vzor MVC příliš vhodný, protože uživatelské prvky již funkcionalitu view a controller kombinují. MVVM je lepší řešení, ale naneštěstí některé uživatelské prvky neposkytují dostatečnou podporu provázání dat (Syromiatnikov a Weyns, 2014). Proto jsme použili návrhový vzor MVP, který se s použitým *Windows Forms* ukázal jako velmi vhodný, a zdrojový kód je tak přehledný a udržovatelný.

3.3.2 Použití v jiných aplikacích

Kromě grafického rozhraní je možné interpret jazyka využít ve formě dynamické knihovny, což se pak hodí ve chvíli, kdybychom chtěli hudební interpret zabudovat například přímo do hry, ve které by pak produkoval za běhu herní hudbu.

3.4 Reprezentace skladby

Při procesu komponování hudby není vytvářen MIDI formát přímo, ale místo toho je vytvářen objekt, který skladbu reprezentuje. Tento objekt je později možné exportovat do formátu MIDI. Výhoda tohoto přístupu je, že při změně hudebního formátu stačí změnit třídu, která funkcionalitu exportu představuje. Díky tomu je v dalších verzích programu možné podporovat export do více hudebních formátů.

Další výhodou reprezentace skladby pomocí objektu vytvořeného pro tento účel, je možnost její serializace a uložení do souboru. Je možné využít binární nebo XML serializaci. Využita je serializace pomocí formátu XML, protože je tím objekt na rozdíl od použití binární serializace přenositelný i na jiné platformy (Maeda, 2012). To je značná výhoda pro možné budoucí rozšiřování aplikace.

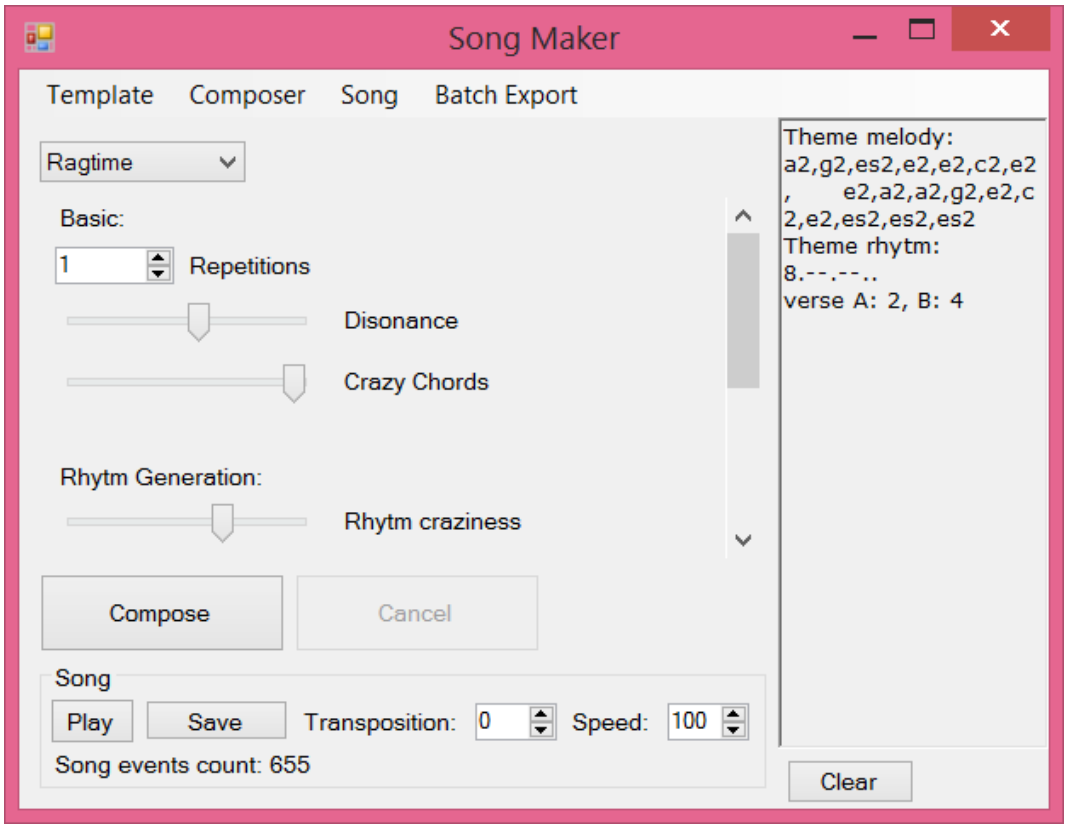
3.5 Platforma

Celá aplikace je naprogramovaná v jazyce C#. Nevýhoda tohoto jazyka spočívá hlavně v tom, že ačkoliv jsou tendence jazyk přenést na unixové systémy, funguje spolehlivě převážně v systému Windows. Přesto jsme se rozhodli tento jazyk použít pro jeho vhodnost k objektově orientovaným a vícevláknovým návrhům a pro jeho podporu firmou Microsoft.

4. Grafické uživatelské rozhraní

Grafické uživatelské rozhraní je důležitou částí aplikace, protože ji zpřístupňuje i lidem bez technického zázemí. Navíc slouží jako interpret jazyka Composing. Tato část neslouží jako uživatelská dokumentace (tu čtenář najde v příloze), ale popisuje nejdůležitější aspekty rozhraní.

Rozhraní je zobrazeno na Obrázku 4.1. Aplikace není lokalizovaná do jiných jazyků než je angličtina.



Obrázek 4.1: Grafické uživatelské rozhraní aplikace Song Maker
Vlevo nahoře si uživatel může nastavit *šablonu*, v tomto případě „Ragtime“, níže může uživatel šablonu přizpůsobit a zkomponovat pomocí tlačítka „Compose“. Zkomponovanou hudbu reprezentuje sekce „Song“ a vpravo nahoře může uživatel vidět výstup z procesu komponování hudby s užitečnými informacemi.

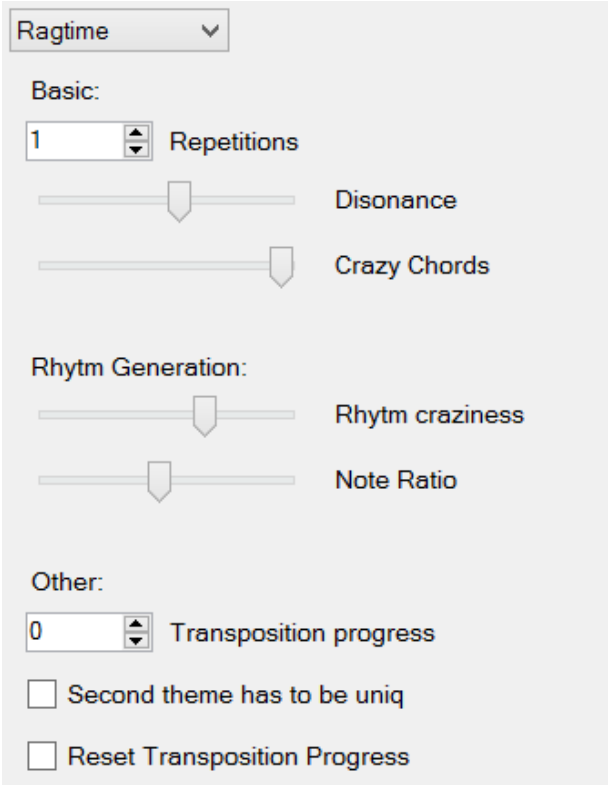
4.1 Výběr šablony

Hlavní funkcí uživatelského rozhraní je interpretace zdrojových kódů jazyka Composing, kterým říkáme šablony. Produkční verze aplikace obsahuje několik vytvořených šablon, pomocí kterých lze komponovat hudební žánry jako např. ragtime a waltz. Součástí grafického rozhraní je také šablona pro zahrání hudebního doprovodu ke skladbě pomocí zadání jejích akordů, čehož lze využit, podobně jako hudebního nástroje, k doprovodu ke zpěvu. V rozhraní si uživatel může potřebnou šablonu zvolit.

4.2 Přizpůsobení šablony

Po zvolení šablony je možné proces komponování hudby přizpůsobit pomocí nastavení parametrů. Tyto parametry uživatel nastaví pomocí, pro tento účel určených, posuvníků, zaškrtávacích políček, textových políček a políček na čísla. Může si tak tedy například nastavit míru harmoničnosti výsledné skladby, její délku, motiv a hudební nástroje, které se ve skladbě použijí. Protože každá šablona odpovídá jinému žánru, má také jiné parametry. S parametry je možné experimentovat a vytvořit si tak hudbu na míru.

Šablona „Ragtime“ například komponuje skladby specifického a osobitého klavírního jazzové stylu, který byl velmi rozšířený na počátku 20. století. Předtím, než bude skladba tohoto žánru zkomponována, uživatel může proces kompozice přizpůsobit nastavením počtu opakování hlavní věty skladby, míry disonance, nastavením komplikovanosti postupů akordů. Dále uživatel může nastavit informace o rytmu a pomocí zaškrtávacího políčka může určit, aby se při každé nové větě skladby transponovala její melodie (viz Obrázek 4.2).



Obrázek 4.2: Přizpůsobení šablony „Ragtime“ v grafickém rozhraní Song Maker.

4.3 Kompozice skladby

Po přizpůsobení šablony je možné skladbu zkomponovat, čímž se vytvoří abstraktní reprezentace skladby v paměti. Poznamenejme, že proces komponování skladby je navržen tak, aby byl v co největší míře nedeterministický, a každá složená skladba je tedy zpravidla jedinečná.

Při kompozici pomocí šablony „Ragtime“ se tak s ohledem na nastavení složí skladba žánru *ragtime* a na výstup se kromě jiného v textové podobě vypíše její hlavní melodie a zvolený rytmus.

4.4 Přehrání skladby a export

Zkomponovaná skladba se převede do formátu MIDI, který je možné přehrát. Již zkomponované skladbě uživatel může nastavit rychlost či ji celou transponovat o zadaný počet půltónů. Pokud se tedy vytvořená skladba uživateli líbí, může s ní ještě dále experimentovat a ještě více ji přizpůsobit svým potřebám.

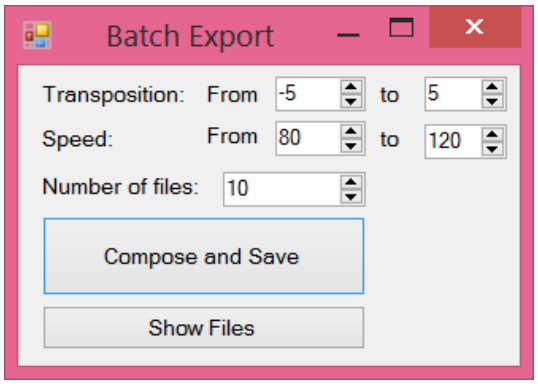
Kromě samotného přehrání je samozřejmě také možné skladbu ve formátu MIDI exportovat do souboru.

4.5 Uložení skladby

Formát MIDI není vhodným formátem pro uložení rozdělané práce, protože například nemá informaci o stisknutých pedálech. Každou skladbu lze proto uložit ve formátu `.song`, který je pro účely Song Makeru vytvořený. Pomocí tohoto formátu je možné vytvořené skladby ukládat do souborů a opětovně je ze souboru do programu načítat. Toho využijeme například ve chvíli, kdy chceme uloženou skladbu jednoduše transponovat nebo změnit její rychlost.

4.6 Dávkový export

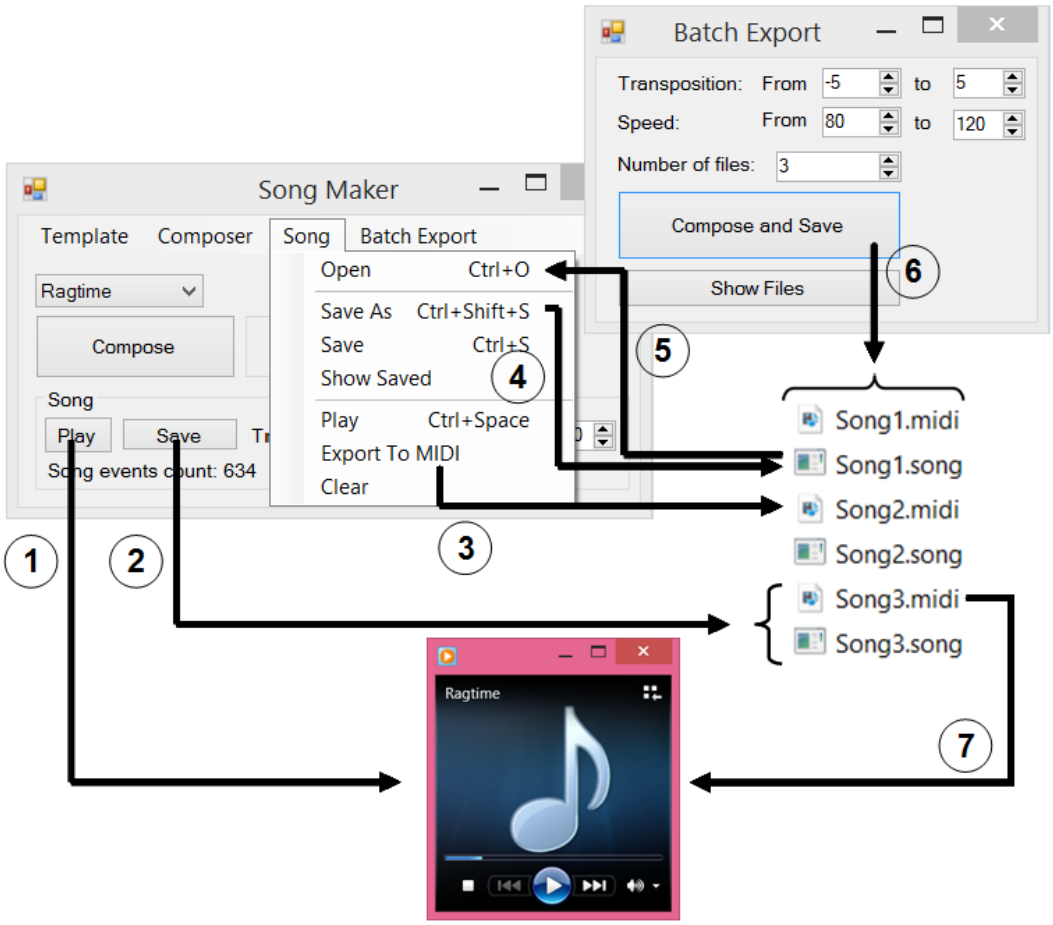
Při používání aplikace se nám velmi osvědčila možnost dávkového exportu. Díky této schopnosti aplikace si může uživatel zvolit šablonu, tu přizpůsobit a potom dávkově zadané množství skladeb zkomponovat, uložit a exportovat. Je pak tedy možné si jednoduše vytvořit dlouhé hudební playlisty a ty pak využít k různým účelům. Při dávkovém exportu je také možnost nastavit, v jakém rozmezí se má určit rychlost skladby a v jakém rozmezí se má skladba před tím, než bude vyexportována do formátu MIDI, transponovat (viz Obrázek 4.3).



Obrázek 4.3: Okénko nastavení dávkového exportu skladeb v aplikaci Song Maker

Obrázek 4.4 znázorňuje práci s uživatelským rozhraním.

Více informací o grafickém rozhraní najdete v uživatelské dokumentaci (viz příloha).

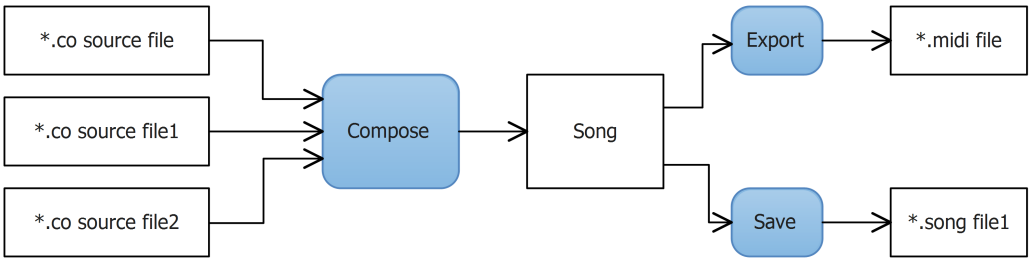


Obrázek 4.4: Uložení, exportování a přehrání skladby v aplikaci *Song Maker* Přímo z GUI je možné zkomponovanou skladbu přehrát (1), jedním kliknutím rychle uložit a exportovat do formátu MIDI (2), exportovat do formátu MIDI (3), uložit jako soubor ve formátu `.song` (4) a tento soubor v rozhraní později otevřít (5), dávkově zkomponovat, uložit a exportovat více skladeb najednou (6) a MIDI soubor je možné otevřít v hudebním přehrávači (7).

5. Hudební jazyk

Jádrem aplikace je vytvořený programovací jazyk *Composing*, který při interpretaci produkuje hudbu. Ačkoli je jazyk interpretovaný a není pomocí něho produkován strojový kód, dá se na proces vytváření hudby dívat jako na překladač. Překladač v tomto případě nepřekládá zdrojové kódy do strojového kódu, ale do skladby, která je reprezentována MIDI souborem nebo souborem formátu `.song`, což je formát, který je pro tento účel vytvořený.

Zdrojové soubory jazyka *Composing* se překladači předloží v podobě cesty ke složce, ve které jsou. Překladač ve složce a jejích podsložkách využije jako zdrojové soubory všechny soubory s koncovkou `.co`. Tyto soubory se překladačem zpracovávají postupně a přednost mají ty s prefixem „*main.*“. Priority *main* souborů se využívá k deklaraci globálních proměnných, které musí být deklarované dříve než metody (viz Obrázek 5.1).



Obrázek 5.1: Proces komponování hudby pomocí jazyka *Composing*

Interpret interpretuje zdrojový kód od začátku. Po interpretaci všech příkazů, které nejsou součástí těla nějaké metody, se zavolá metoda `main()`, pokud existuje.

5.1 Standardní metody pro práci s hudbou

Pro skládání skladeb se využívají standardní metody, které jsou v jazyce nadefinovány. Seznam těchto metod poskytne příkaz `help()`. Zde je několik prvních řádků výstupu tohoto příkazu:

```
=== HELP ===
All standard functions available follows.
Use help(function) for more info (e.g. help('help')).
===

at(...)
        Returns array element by index.

count(...)
        Returns number of array elements.

...
```

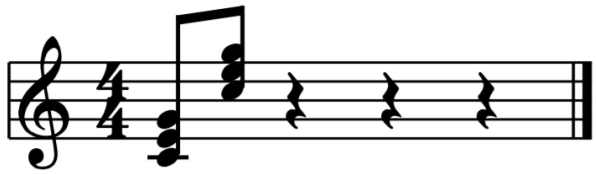
Metoda `melody(...)` slouží k vytvoření melodie. Pro nápovědu k této metodě využijeme příkaz `help("melody")`:

```
=== HELP ===
melody(...)
        Sets or gets stored melody.

        USAGE: melody(), melody(melody)
        RETURN: new melody
        EXAMPLE: melody('c, e, c c2')
```

Melodie je specifikována pomocí řetězce, kde mezerou jsou odděleny tóny v rámci jednoho akordu, které se hrají najednou, a čárkou jsou odděleny tóny či akordy, které se hrají časově postupně. Kromě zvolení melodie je třeba před přidáním tónů do skladby specifikovat i rytmus pomocí metody `rhythm(...)`. Když jsou melodie i rytmus nastaveny, můžeme specifikované tóny vytvořit a přidat do skladby pomocí metody *add()*:

```
melody("C E G, C2 E2 G2");
rhythm("8--"); /* 2 osminové noty */
add();
```

Pro přeložení zdrojového kódu vytvoříme složku `Demo` a v ní soubor `main.co` s třemi řádkami zdrojového kódu výše. V GUI *SongMaker* otevřeme šablonu Demo a klikneme na tlačítko *Compose*. Následně bude zkomponovaná hudba (viz Obrázek 5.2).



Obrázek 5.2: Hudba vytvořená pomocí jazyka *Composing*

V jazyce *Composing* existuje množství standardních metod, které vytváření hudby usnadní. Skladbu, kterou jsme výše vytvořili, můžeme popsat tak, že se skládá ze dvou akordů C-dur, kde druhý je o oktávu (12 půltónů) výš než první. Takový popis můžeme zapsat i pomocí jazyka:

```
tones = chord("C"); /* tóny C E G */
m = format("{0},{1}", tones, transpose(tones, 12));
melody(m);
rhythm("8--"); /* 2 osminové noty */
add();
```

Použitá metoda `chord(...)` parsuje název akordu a vrátí seznam tónů, ze kterých se skládá. Metoda `format(...)` vloží hodnoty druhého a případně dalších parametrů na místa „`{0}`“, „`{1}`“, ... v prvním parametru. Použití metody `format` je podobné stejnojmenné metodě v jazyku C#.

V jazyce je možné definovat a volat vlastní metody:

```
part("C");
part("G");
part("G7");
part("C");


def part(ch){
        tones = chord(ch); /* tóny C E G */
        m = format("{0},{1}", tones, transpose(tones, 12));
        melody(m);
        rhythm("8--"); /* 2 osminové noty */
        add();
        play();
}
```

Použitý příkaz `play()` změří délku rytmu uloženého pomocí metody `rhythm(...)` a o tuto délku se přesune v komponované skladbě dále. Hudba se tak skládá po jednotlivých částech od začátku skladby do konce.

Také můžeme využívat ovládací příkazy známé z jiných jazyků, jako např. `if`, `for`, `while`, `do`, `break` a `continue`:

```
chords = "C G G7 C";

for(i = 0; i < count(chords); i += 1){
        ch = at(chords, i);
        part(ch);
}

def part(ch){
        tones = chord(ch); /* tóny C E G */
        m = format("{0},{1}", tones, transpose(tones, 12));
        melody(m);
        rhythm("8--"); /* 2 osminové noty */
        add();
        play();
}
```

Metody `count(...)` a `at(...)` jsou jedny z metod, které přidávají do jazyka *Composing* funkcionalitu polí. Pole je reprezentováno řetězcem a jeho prvky, pokud není pomocí volitelného parametru nastaveno jiné chování, jsou oddělené libovolným počtem mezer. Použitá metoda `count(array)` vrátí velikost pole `array` a metoda `at(array, i)` vrátí i-tý prvek pole `array`.

5.2 Nedeterminizmus a definice metod

Aby možných skladeb zkomponovaných pomocí jedné šablony mohlo být co největší množství, očekáváme při komponování hudby velký prvek náhodnosti. Jazyk je proto navržen tak, aby jeho interpretace byla co nejméně deterministická. Toho je docíleno pomocí možnosti definovat jednu metodu několikrát. Při zavolání takové metody se zavolá náhodná z nich.

Navíc můžeme pro zkrácení kódu při definování více těl jedné metody napsat hlavičku metody (klíčové slovo `def`, název metody a parametry) jen jednou a následovat ji několika bloky definující různá těla metody. To je ekvivalentní tomu, pokud bychom definovali stejnou metodu vícekrát. Pokud se blok, který je tělem metody, skládá jen z jednoho příkazu, můžeme dokonce vynechat složené závorky.

Tak si můžeme například definovat několikrát metodu `chords()`, která vrátí seznam akordů budoucí skladby:

```
def chords()
        return "C G G7 C";
        return "C F G C";
        return "Am Dm Am E7";
        return "C D7 G7 Am Fdim7 Cdim7 G7 C";
```

Metodu pak můžeme využít k nedeterministickému zvolení postupu akordů uložením do proměnné `chords`:

```
chords = chords();

for(i = 0; i < count(chords); i += 1){
        ch = at(chords, i);
        part(ch);
}
```

5.3 Simulace levé i pravé ruky

Pokud hraje klavírista nějakou skladbu, používá k tomu levou a pravou ruku, pro něž je notový zápis zapsán v notách odděleně. Levou rukou se zpravidla hraje tzv. doprovod a pravou rukou tzv. melodie skladby.

Tímto se můžeme inspirovat a použít princip oddělení práce levé a pravé ruky v kompozici. Metodu pro zahrání levé ruky označíme `left` a metodu pro zahrání pravé ruky `right`. Aby skladba zněla dobře, obě ruce by se neměly příliš mísit. Toho docílíme tím, že tóny v levé ruce posuneme o dvě oktávy níže.

5.3.1 Harmoničnost

Při hraní melodie pravou rukou je třeba dbát na to, aby melodie zněla v závislosti na doprovodu v pravé ruce harmonicky. Toho je možné docílit například tak, že pravá ruka bude hrát jen tóny, ze kterých se skládá akord hraný v levé ruce, a ty od nich posunuté o násobky oktáv. Toho docílí standardní metoda

`spread(...)`, která vrátí seznam všech tónů v určitém rozsahu, které jsou harmonické se zadanými tóny. Následující ukázka k akordům přidá melodii hranou pravou rukou pomocí náhodných šestnáctinových not, které jsou harmonické s odpovídajícími akordy.

```
chords = chords();
print("Chords: {0}", chords);

for(i = 0; i < count(chords); i += 1){
        ch = at(chords, i);
        part(ch);
}

def chords()
        return "C G G7 C";
        return "C F G C";
        return "Am Dm Am E7";
        return "C D7 G7 Am Fdim7 Cdim7 G7 C";

def part(ch){
        left(ch);
        right(ch);
        play();
}

def left(ch){
        tones = chord(ch);
        m = format("{0},{1}", tones, transpose(tones, 12));
        m = transpose(m, -24);
        melody(m);
        rhythm("8--");
        add();
}

def right(ch){
        tones = chord(ch);
        m = spread(tones, "C2", "C4");
        m = shuffle(m);
        m = replace(m, " ", ",");
        m = transpose(m, -12);

        melody(m);
        rhythm("16----");
        add();
}
```

Metoda `shuffle(...)` zamíchá pole a metoda `replace(...)` nahradí výskyt podřetězce v řetězci jiným řetězcem. V ukázce výše je funkcionalita levé a pravé ruky pro přehlednost oddělena v metodách `left` respektive `right`. Za pozornost stojí rozdíl v chování metod `add(...)` a `play(...)`. Metoda `add(...)` přidá na

aktuální pozici v komponované skladbě úsek tónů, které jsou specifikované pomocí nastaveného rytmu a melodie, a aktuální pozici nemění. Metoda `play(...)` změří velikost tohoto úseku a přesune aktuální pozici dále. Kód uvedený výše přidává tóny do výsledné skladby tím způsobem, že na aktuální pozici přidá úsek pro levou ruku, poté na stejnou pozici úsek pro pravou ruku, pak se pomocí příkazu `play` přesune ve skladbě dále, a takto celý proces opakuje, dokud skladbu nedokončí.

5.4 Atribut `weight` definice metody

Při volání metody s více definicemi, je zvolená náhodná z nich z rovnoměrného rozdělení. Pokud chceme nějakou definici upřednostnit před ostatními, můžeme toho docílit pomocí atributu `weight`, který změní váhu definice metody. Výchozí váha metody je 1. Váha 2 znamená, že pravděpodobnost, že se definice metody využije, je dvakrát větší než pravděpodobnost, že se využije definice s váhou 1. Pokud je váha rovna 0, definice se nikdy nepoužije. Místo čísla je také možné napsat výraz. Atributy jsou označené hranatými závorkami a píší se před definici těla metody.

Pro větší nedeterminističnost programu můžeme pro pravou ruku vybrat vždy náhodný rytmus:

```
def rh()
        return "16----";
        return "16-.--";
        return "32--------";
        return "32--------";
        [weight 2]
        return "32-..-..-.";
```

A pak můžeme metodu využít k vytvoření rytmu pro pravou ruku:

```
rhythm(rh());
```

Rytmus `32-..-..-.` se v uvedeném příkladu použije v průměru dvakrát častěji než každý z ostatních rytmů.

5.5 Opakování skladby

Jednou z častých pomůcek při komponování hudby je využití repeticí, neboli opakování úseků skladby či opakování skladby celé. V jazyce `Composing` opakování docílíme například pomocí cyklů. V následující ukázce k opakování využíváme cyklu *for*:

```
REPETITIONS = 4;

chords = chords();
print("Chords: {0}", chords);
```

```
for(ii = 0; ii < REPETITIONS; ii += 1)
for(i = 0; i < count(chords); i += 1){
        ch = at(chords, i);
        part(ch);
}
```

5.6 Argumenty programu a ovládací prvky k přizpůsobení šablony

Na předchozí ukázce je pomocí globální proměnné `REPETITIONS` nastaven počet opakování skladby a to na hodnotu 4, kterou uživatel bez zásahu do kódu šablony nemůže změnit. Abychom umožnili uživateli před kompozicí počet opakování změnit, můžeme využít tzv. **vstupních argumentů**.

Grafické uživatelské rozhraní *SongMaker*, které slouží jako interpret jazyka *Composing*, umožňuje předávat programům v jazyce vstupní argumenty. Argumenty jsou v grafickém rozhraní představeny ovládacími prvky (viz Obrázek 4.2) a ve zdrojovém kódu jazyka reprezentovány jako inicializované globální proměnné.

Jednou ze zajímavostí při tvoření vlastní šablony je, že každý skladatel může přímo popsat způsob, jakým uživatel má zadat vstupní argumenty, a to i vizuálně. Ovládací prvky pro zvolenou šablonu se popíší pomocí jazyka se stejnou syntaxí jako *Composing*, který má však místo standardních metod pro práci s hudbou metody na vytváření uživatelských prvků. Překlad tohoto jazyka funguje stejně jako překlad jazyka *Composing* s tou výjimkou, že jeho zdrojové soubory mají koncovku `.args`. Ve složce *Demo* tak vytvoříme soubor `main.args` s obsahem:

```
label('Basic setting:'); br();
numeric('REPETITIONS','Repetitions',2,1,8);
```

Použitá metoda `label` způsobí vykreslení popisku, metoda `br` zalomí řádek a metoda `numeric` s použitými parametry výše vytvoří ovládací prvek na zadání celého čísla v rozmezí 1 až 8 s výchozím číslem 2. Vedle tohoto prvku se vykreslí popisek „Repetitions“ a při spuštění procesu kompozice se jeho hodnota nastaví jako hodnota proměnné `REPETITIONS`, která je použita v šabloně.

5.6.1 Podmínkový atribut definice metody

Kromě váhy definice metody je možné využít *podmínkového atributu*, který nastaví, že se definice použije jen v případě, když je hodnota výrazu podmínky pravdivá.

To usnadní práci při přizpůsobení šablony pomocí vstupních argumentů. Do ukázkové šablony přidáme vstupní číselný argument `DISSONANCE` k určení disonance výsledné hudby a pravdivostní argument `MINOR_CHORDS`, pomocí kterého uživatel může povolit či zakázat mollové postupy akordů. Mollové postupy jsou takové postupy, které zní „smutně“.

Soubor `main.args`:

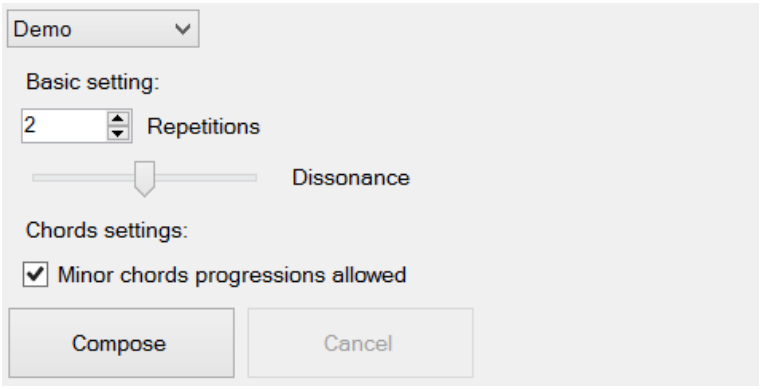
```
label('Basic setting:'); br();
numeric('REPETITIONS','Repetitions',2,1,8); br();
slider('DISSONANCE','Dissonance',5,0,10); br();

label('Chords settings:'); br();
check('MINOR_CHORDS', "Minor chords progressions allowed", true); br();
```

Soubor `main.co`:

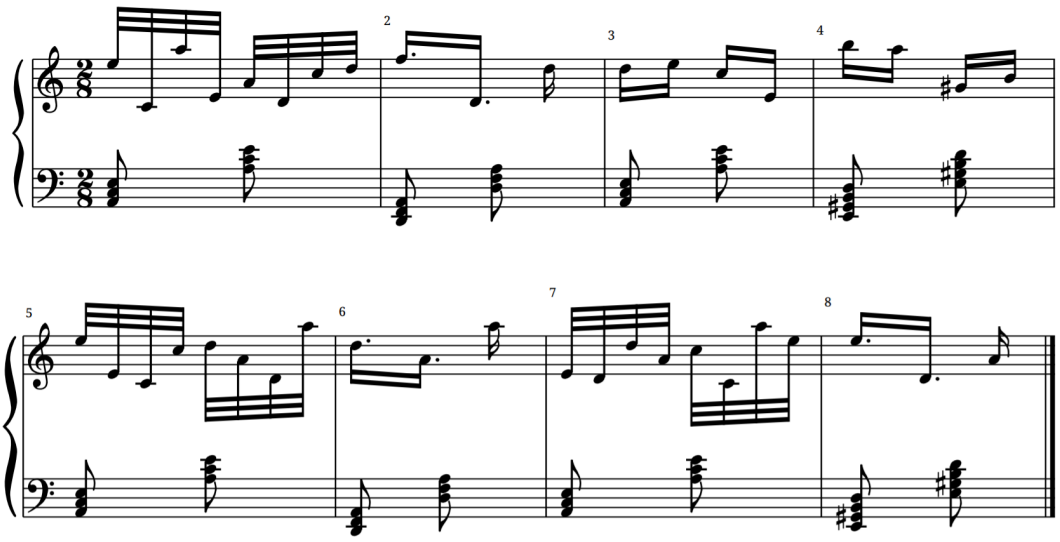
```
chords = chords();
print("Chords: {0}", chords);

for(ii = 0; ii < REPETITIONS; ii += 1)
for(i = 0; i < count(chords); i += 1){
        ch = at(chords, i);
        part(ch);
}

def chords()
        return "C G G7 C";
        return "C F G C";
        [MINOR_CHORDS]
        return "Am Dm Am E7";
        [DISSONANCE > 6]
        return "C D7 G7 Am Fdim7 Cdim7 G7 C";

def part(ch){
        left(ch);
        right(ch);
        play();
}

def left(ch){
        tones = chord(ch);
        m = format("{0},{1}", tones, transpose(tones, 12));
        m = transpose(m, -24);
        melody(m);
        rhythm("8--"); /* 2 osminové noty */
        add();
}

def right(ch){
        tones = chord(ch);

        if(DISSONANCE > 2)
                tones += " " + "A";
        if(DISSONANCE > 4)
                tones += " " + "D";
        if(DISSONANCE > 9)
                tones += " " + "Eb";
```

```
        m = spread(tones, "C2", "C4");
        m = shuffle(m);
        m = replace(m, " ", ",");
        m = transpose(m, -12);

        melody(m);
        rhythm(rh());
        add();
}

def rh()
        return "16----";
        return "16-.--";
        return "32--------";
        return "32--------";
        [weight 2]
        return "32-..-..-.";
```

Vytvořenou šablonu Demo můžeme otevřít v GUI *SongMaker* (viz Obrázek 5.3) a zkomponovat skladbu. Notový zápis skladby je zobrazen na Obrázku 5.4.



Obrázek 5.3: Ovládací prvky na přizpůsobení šablony Demo
Před kompozicí skladby uživatel může nastavit počet opakování, míru disonance a určit, zda ve skladbě mohou být mollové („smutné“) postupy akordů.



Obrázek 5.4: Skladba složená pomocí šablony Demo napsané v jazyce *Composing*
Tato klavírní skladba začíná zahraním rychlejší melodie pravou rukou doprovázené mollovými akordy levou rukou, a přechází k pomalejší melodii a durovým akordům, které jsou zakončené akordy E7, což má vzbuzovat napětí. Podobný motiv se opakuje a je zahrána melodie, která je zpočátku rychlá, poté se zpomalí, při přechodu na durové akordy se melodie opět zrychlí a skladba je zakončena opět zahráním akordu E7.

6. Použité technologie

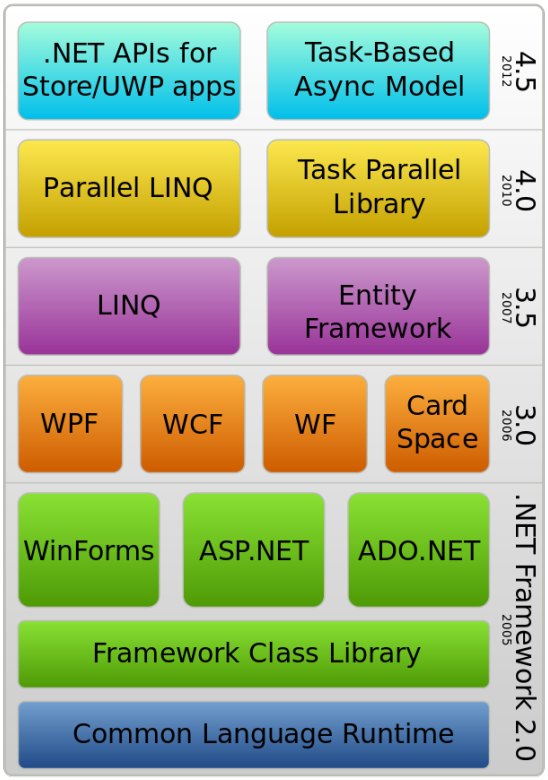
6.1 C#

Aplikace je napsaná v jazyku C# a využívá framework .NET 4.5.

6.1.1 .NET

Framework .NET je vyvíjený firmou Microsoft a funguje tedy nejlépe na systému Windows. Verze 4.5 vznikla v roce 2012 a je podporována na operačních systémech Windows Vista a novějších verzích Windows (Brandon Bray, 2012). .NET Framework 4.5 využívá Common Language Runtime 4.0 a je podporován také v systémech Windows Server 2008, Server 2008 R2, Server 2012 a Server 2012 R2. Aplikace také poběží na systémech kde je nainstalovaný .NET Framework 4.6 (MSDN:.NET Framework Versions and Dependencies).

Aplikace využívá .NET Framework 4.5 pro jeho funkcionalitu, která se týká *asynchronous tasks* a *futures and promises* (viz Obrázek 6.1). Konkrétně je díky této verzi frameworku v aplikaci task-based model umožňující spuštění interpretu a komponování hudby asynchronně a umožňující přerušení tohoto procesu.



Obrázek 6.1: Komponenty .NET Frameworku.
V horní části obrázku vidíme komponentu „Task-Based Async Model“, která je v aplikaci využita. Obrázek použit z dostupného zdroje pod licencí CC BY-SA 3.0 `https://commons.wikimedia.org/wiki/Category:.NET#/media/File:DotNet.svg`.

Kromě *asynchronous tasks* aplikace využívá komponentu LINQ (Marguerie a kol., 2008), která pomáhá v práci s kolekcemi, a XML Serializaci[1], pomocí které reprezentuje složenou hudbu.

6.2 Windows Forms

Windows Forms je třída na grafické uživatelské rozhraní (GUI), která je součástí použitého .NET Frameworku. Aplikace tuto třídu využívá na implementací uživatelského rozhraní Song Maker.

Třída představuje abstrakci nad nativním Windows User Interface Common Controls a zapouzdřuje tím tak Windows API.

S uživatelským rozhraním se pomocí této třídy pracuje pomocí objektového systému ovládacích prvků. Třída definuje základní nabídku ovládacích prvků, které se běžně používají v operačním systému Windows, ale lze také přidávat vlastní prvky, kterým se říká Custom Controls.

S využitím Custom Controls jsou v aplikaci Song Maker vytvořené prvky na přizpůsobení šablon.

6.3 MIDI

MIDI (Musical Instrument Digital Interface) je standard z roku 1981 používaný k hudebním účelům, který definuje komunikační protokol dovolující elektronickým hudebním nástrojům, počítačům a dalším zařízením v reálném čase komunikovat pomocí sériového rozhraní. Standard také definuje souborový formát pro popis hudby, čehož se v této aplikaci využívá.

6.3.1 MIDI události a MIDI zprávy

Komunikace hudebních zařízení je provedena pomocí tzv. MIDI událostí na úrovni bytů. Každá MIDI událost se skládá z jednoho bytu, který specifikuje čas události pomocí tzv. MIDI tiků a dalších bytů, které specifikují akci, která se má provést, a nesou označení MIDI zpráva. MIDI zpráva pak sestává z 1. bytu, který určuje typ zprávy, a z případných dalších datových bytů[2].

6.3.2 Kanály

Některé typy MIDI událostí mohou mít specifikovaný jeden z 16 kanálů (channel). Díky tomu se pak při MIDI zprávě o změně hudebního nástroje nezmění hudební nástroj všech, od dané chvíle hraných, tónů, ale jen takových, které jsou zahrané ve stejném kanálu.

[1] `https://msdn.microsoft.com/en-us/library/182eeyhh(v=vs.110).aspx`

[2] `http://www.recordingblogs.com/sa/Wiki/topic/Musical-Instrument-Digital-Interface-MIDI`

6.3.3 General MIDI

Specifikace formátu MIDI je do velké míry abstraktní a dodefinovává ho standard General MIDI (GM) z roku 1991. GM specifikuje hudební nástroje, které odpovídají na konkrétní MIDI zprávy a v aplikaci ho využíváme. SMF je tedy souborový formát k uložení formátu MIDI, GM je upřesnění formátu MIDI.

Komunikační protokol, který GM dodefinovává, zaručuje, že odlišné elektronické hudební nástroje budou vzájemně lépe komunikovat (tedy že například po stisku klávesy na MIDI klavíru zahraje připojený MIDI modul správný tón).

Standard GM na MIDI zařízení klade minimální požadavky. GM zařízení musí splňovat:

- Podporuje 128 hudebních nástrojů. Definuje zvuky těchto hudebních nástrojů.
- Minimálně 24 zvuků může být hráno najednou. Z toho 16 zvuků nástrojů, které mohou hrát melodii, a 8 bicích.
- Reakce na sílu zahrání tónu.
- Podpora všech 16ti kanálů najednou, kde kanál s číslem 10 je rezervován pro bicí hudební nástroje. Na kanálu 10 je pro některé tóny zvolen nějaký bicí nástroj, který se zahraje místo tónu.
- V jednom kanálu může být hráno více tónů najednou.

6.3.4 Standard MIDI File

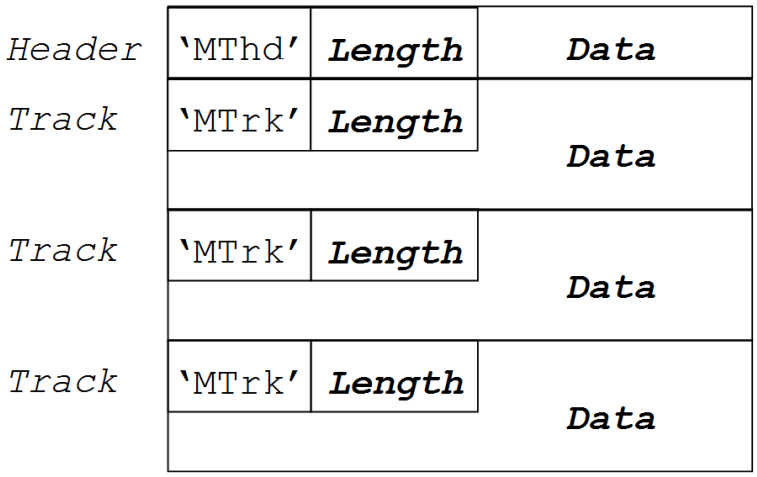
Ve standardu General MIDI je kromě výše uvedených požadavků specifikován i souborový formát Standard MIDI File (SMF). Formátu SMF v aplikaci využíváme k exportování a přehrání složených skladeb. SMF je organizován do několika částí. Hlavičková (header) část obsahuje informace o celé skladbě a stopy (tracks) obsahují MIDI události (viz Obrázek 6.2). V naší aplikaci k popisu skladby používáme jen jednu stopu.

MIDI soubory mají koncovku `.midi` případně `.mid`.

6.3.5 MIDI syntezátor

K přehrání MIDI souboru je třeba tzv. MIDI syntezátoru, který interpretuje MIDI zprávy a syntetizuje tak výsledný zvuk. Ten může být např. zabudovaný na zvukové kartě počítače nebo vytvořený softwarově.

I když MIDI syntezátory odpovídají standardu GM, zvuky hudebních nástrojů zpravidla nebývají pro různé syntezátory stejné. Tím pádem výsledný zvuk zkomponované a vyexportované skladby aplikací Song Maker závisí na tom, jaký syntezátor je použit. Velmi dobrých výsledků jsme dosáhli při přehrání skladby na Mac OS Yosemite. V systému Android Jelly Bean se naopak syntezátor ukázal jako ne příliš kvalitní.



Obrázek 6.2: Členění souborového formátu SMF
Standard MIDI File je členěn na hlavičkovou („Header“) část a jednu nebo více stop („Track“). Hlavička je uvozena řetězcem „MThd“, po kterém následuje délka hlavičky a data. Stopa začíná řetězcem „MTrk“, který je následovaný délkou stopy a daty.

6.3.6 C# MIDI Toolkit

Aplikace využívá pro podporu MIDI DLL knihovny C# MIDI Toolkit[3]. Jejím autorem je Leslie Sanford a je pod licencí MIT. Knihovna definuje rozhraní s formátem MIDI v jazyce C#.

6.4 GPLEX

Gardens Point LEX[4] (GPLEX) je scanner generátor, který slouží k produkci lexikálních scannerů napsaných v jazyce C# využívající .NET Framework 2.0. Více o principu lexikálních scannerů v Sekci 7.2.1. GPPLEX je šířen pod BSD-like licencí. Tento generátor používáme na generování lexikálního scanneru zdrojových kódů jazyka Composing.

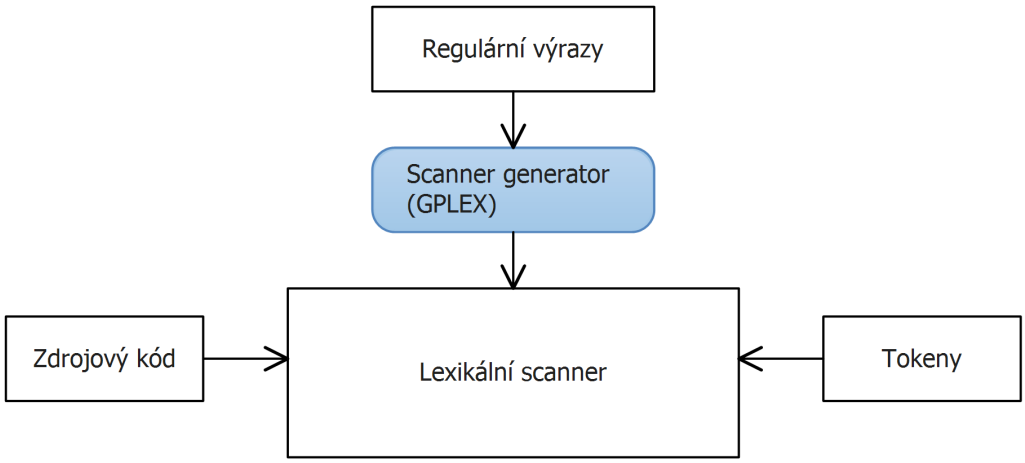
GPLEX generuje lexikální scannery, které pracují pomocí konečných stavových automatů. Vygenerované automaty jsou optimalizované tak, že mají minimální počet stavů.

Tento generátor přijímá popis scanneru podobné POSIXové specifikaci LEXu a produkuje C# soubor (viz Obrázek 6.3). GPLEX má navíc podporu pro unicode, čehož využíváme. Následuje ukázka lexikální analýzy relačních operátorů v jazyce *Composing*. Relační operátory jsou pomocí regulárních výrazů nalezeny lexikálním scannerem ve zdrojovém kódu jazyka *Composing*. Když tento lexikální scanner rozezná relační operátor, reprezentuje ho pomocí tokenu `OPER_REL`. Typ operátoru uloží pomocí atributu `oper_bin`, čímž token specifikuje.

```
\<    yylval.oper_bin = Operator.Binary.LT;
      return (int)Tokens.OPER_REL;
```

[3] https://www.codeproject.com/Articles/6228/C-MIDI-Toolkit
[4] https://gplex.codeplex.com

```
\<=  yylval.oper_bin = Operator.Binary.LE;
     return (int)Tokens.OPER_REL;
!=   yylval.oper_bin = Operator.Binary.NE;
     return (int)Tokens.OPER_REL;
==   yylval.oper_bin = Operator.Binary.EQ;
     return (int)Tokens.OPER_REL;
>=   yylval.oper_bin = Operator.Binary.GE;
     return (int)Tokens.OPER_REL;
>    yylval.oper_bin = Operator.Binary.GT;
     return (int)Tokens.OPER_REL;
```



Obrázek 6.3: Schéma lexikálního generátoru GPLEX
Pomocí regulárních výrazů se popíše lexikální scanner, který je následně vygenerovaný pomocí GPLEXu. Vygenerovaný scanner pak čte zdrojový kód programovacího jazyka a produkuje proud tokenů, které jsou použity dále při překladu.

6.5 GPPG

Gardens Point Parser Generator[5] (GPPG) je program, který generuje parser napsaný v jazyce C#. Tento parser využíváme ke zpracování tokenů získaných z lexikální analýzy a k postavení abstraktního syntaktického stromu. Více o funkci parseru a tokenů v Sekci 7.2.1.

Generovaný parser je popsaný formální gramatikou pomocí YACC-like syntaxe. Následuje ukázka přepisovacích pravidel pro cyklus for a while jazyka Composing v YACC-like syntaxi. Výrazy `for` a `while`, které jsou na levé straně kódu níže, představují neterminály. Tyto neterminály jsou přepisovány posloupností, která se skládá z dalších neterminálů a terminálů, které odpovídají tokenům.

```
/* for(;;) <block> */
for : FOR LPAR runnable SEMICOLON valuable SEMICOLON
```

[5] `https://gppg.codeplex.com/`

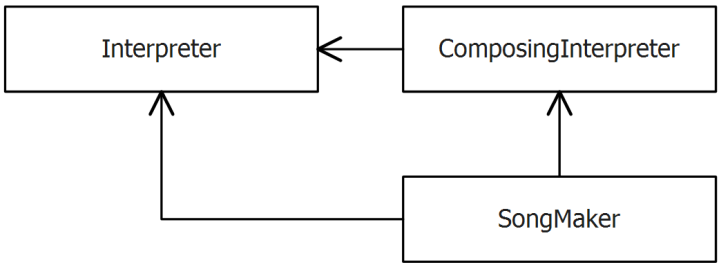
```
      runnable RPAR block
      { $$ = new For($3,$5,$7,$9, @$); } ;

/* while() <block> */
while : WHILE LPAR valuable RPAR block
        { $$ = new While($3, $5, @$); } ;
```

Generátor na základě YACC-like souboru vytvoří kód v jazyce C#, který parser představuje. GPPG generuje LALR(1) parsery (Look-Ahead Left-to-Right), které parsují zdola nahoru (bottom-up).

7. Návrh aplikace

7.1 Přehled

Zdrojový kód aplikace, je rozdělen na 3 hlavní komponenty (viz Obrázek 7.1).



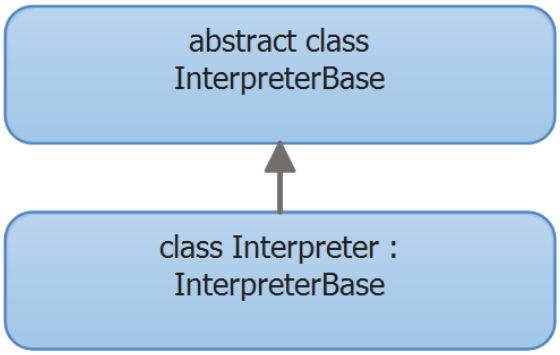
Obrázek 7.1: Závislosti hlavních komponent aplikace

- **`Interpreter`** slouží k interpretaci zdrojových kódů jazyka nezávislého na hudbě. Definuje například syntaktické výrazy (např. *if* a *for*) a standardní metody.
- **`ComposingInterpreter`** rozšiřuje `Interpreter` o práci s hudbou a umožňuje ukládání a exportování skladeb do formátu MIDI.
- **`SongMaker`** vytváří uživatelské okenní rozhraní. Využívá `ComposingInterpreter` ke komponování skladeb a `Interpreter` na vytvoření ovládacích prvků sloužících ke přizpůsobení šablon.

Následuje programátorská dokumentace, která postupně popisuje komponenty aplikace.

7.2 Interpreter

Interpreter je realizovaný veřejnou třídou `Interpreter`. Tato třída je potomkem `InterpreterBase` (viz Obrázek 7.2).

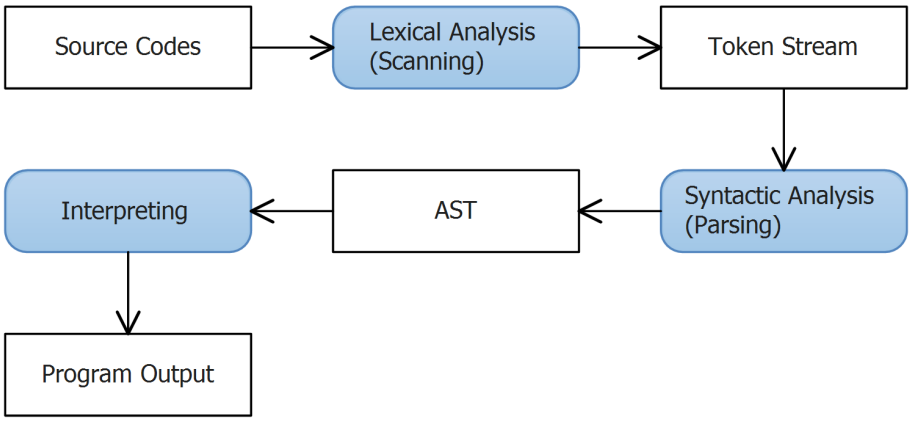


Obrázek 7.2: Závislost tříd `Interpreter` a `InterpreterBase`

Abstraktní třída `InterpreterBase` využívá lexikální a syntaktické analýzy a definuje syntaxi jazyka. Její potomek `Interpreter` dodefinovává standardní metody jazyka.

7.2.1 Překlad

Překlad zdrojových kódů jazyka funguje v několika fázích (viz Obrázek 7.3).

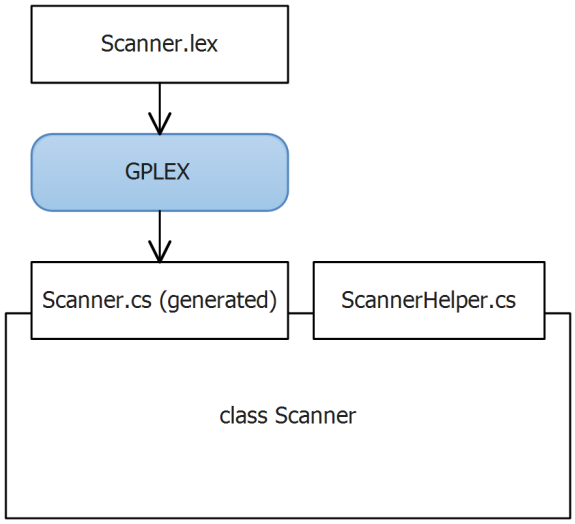


Obrázek 7.3: Data flow při překladu jazyka pomocí třídy `Interpreter`
Ze zdrojového kódu je nejprve vytvořen proud tokenů, který je za pomoci syntaktické analýzy použit k vytvoření AST. Program reprezentovaný AST je interpretován, což má za následek jeho výstup.

Lexikální analýza

Složka se zdrojovými kódy jazyka v textovém zápisu je nejdřív zpracována pomocí lexikálního scanneru. Lexikální scanner je reprezentován třídou `Scanner`, která je vytvořena ze souboru `Scanner.lex` za použití generátoru scannerů

GPLEX, jehož výstupem je soubor `Scanner.cs`. V tomto souboru je definovaná třída `Scanner`. Třída je v jazyce C# označovaná jako *partial* a její druhá část je v souboru `ScannerHelper.cs`, který obsahuje pomocné metody pro scanner (viz Obrázek 7.4).



Obrázek 7.4: Vytvoření scanneru pro lexikální analýzu

`Scanner` postupně čte vstupní soubory, které navíc vhodně uspořádá, a produkuje proud tokenů. Tokeny reprezentují lexikální elementy jazyka jako jsou např. klíčová slova a literály. Při produkování tokenů scanner ignoruje komentáře a hlásí případné lexikální chyby. Následuje ukázka lexikální chyby, kterou program vypíše na standardní výstup. Tato chyba vznikla kvůli neočekávanému znaku tečky („.“) na třetím řádku zdrojového kódu:

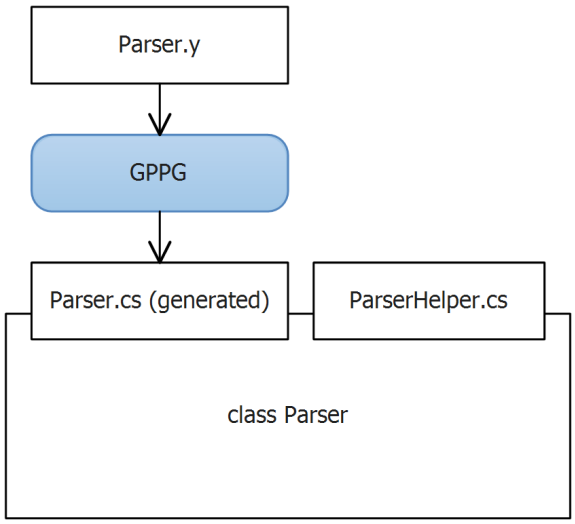
```
Error "Lexical error: Unexpected char" in source.txt:3:0
  > .
```

Syntaktická analýza

Proud tokenů je zpracován parserem reprezentovaným třídou `Parser`, která je vytvořena za pomoci parser generátoru *GPPG* ze souboru `Parser.y`. Soubor `ParserHelper.cs` definuje pomocné metody pro `Parser` (viz Obrázek 7.5).

Cílem syntaktické analýzy je odhalit syntaktické chyby a vytvořit AST (Abstract Syntax Tree). Následuje ukázka syntaktické chyby vypsané programem při špatném formátu cyklu *for* na pátém řádku zdrojového kódu. Pátý řádek má tvar `for(true){}`.

```
Error "Syntax error, unexpected BOOL" in source.txt:3:4
  > true
Errors occured
```



Obrázek 7.5: Vytvoření parseru pro syntaktickou analýzu

Abstract Syntax Tree

Abstract Syntax Tree (AST) je stromová reprezentace syntaktické struktury zdrojového kódu (viz Obrázek 7.6).

Uzly stromu jsou představené třídami, které implementují rozhraní `INode` a mohou implementovat i další rozhraní, které určí jejich sémantický typ. Tedy například uzly `MethodCall`, `BinaryOperator`, `IntLiteral`, `UnaryOperator` a `FloatLiteral` implementují rozhraní `IValuable`, které o nich říká, že z nich může být při interpretaci získána hodnota. Uzly, které implementují toto rozhraní, pak mohou být ve stromové struktuře např. jako argumenty metody `MethodCall`.

Interpretace AST

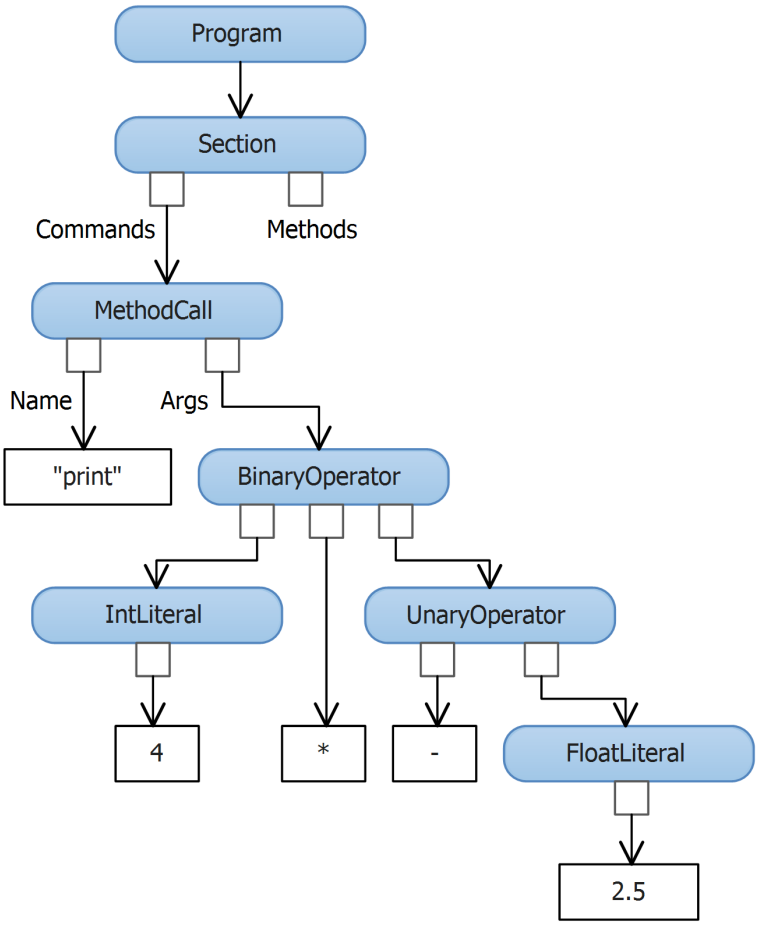
Při interpretaci stromové struktury se nabízí možnost nějakým způsobem interpretovat každý uzel. Kdyby se každý uzel uměl sám interpretovat, mohl by pak spustit interpretaci svých přímých potomků a k interpretaci celého stromu by pak stačilo spustit interpretaci kořene.

Standardně je možné interpretaci sama sebe z programátorského hlediska vyřešit vytvořením zvláštní třídy pro každý uzel. AST je pak možné složit z těchto tříd, které v sobě budou mít nadefinovanou funkcionalitu pro interpretování sebe sama.

Ukázalo se však, že definovat funkcionalitu interpretace přímo v uzlech je nevhodné. Důvodem je, že je třeba při takovém postupu využívat v syntaktické analýze, pomocí které se buduje AST, třídy, které již řeší funkcionalitu interpretace. K vytvoření AST by však znalost o interpretaci být potřebná neměla (viz Obrázek 7.3).

Jak také uvádí Jeanmart a kol. (2009), kód se při výše uvedeném přístupu stává složitějším na údržbu a modifikace. Proto jsme pro interpretaci abstraktního syntaktického stromu zvolili *visitor pattern*.

Visitor Pattern je návrhový vzor, který umožňuje rozšiřovat funkce objektu bez nutnosti modifikovat jeho třídu. Při použití tohoto vzoru pro každou novou



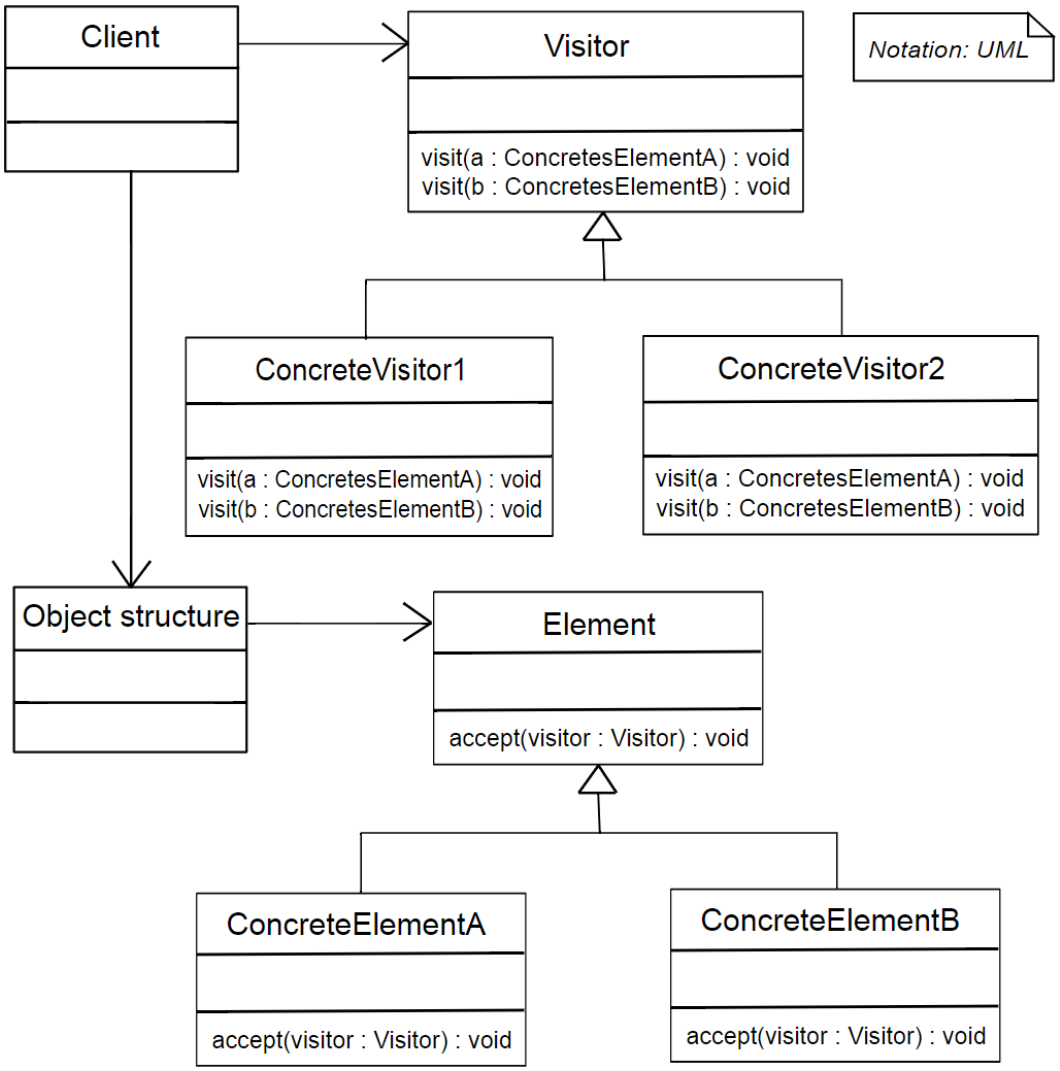
Obrázek 7.6: Příklad Abstract Syntax Tree vytvořený pomocí třídy `Interpreter`

akci, kterou chceme přidat k původní množině tříd, vytvoříme novou třídu, která představuje *návštěvníka*. Obvyklé pojetí implementace návštěvníka je takové, že na třídě, jejíž funkcionalitu chceme modifikovat, vytvoříme metodu `Accept`, která návštěvníka „přijme“ (viz Obrázek 7.7). My však v optimalizaci přehlednosti zdrojového kódu jdeme ještě dál a využíváme *.NET Reflection*.

Díky *.Net Reflection* je možné vytvořit tzv. *dynamic dispatcher*, díky kterému může program za běhu vybrat, které přetížení metody zvolí. Vytvoříme tak díky tomu návštěvníka uzlů AST, aniž bychom pro to museli třídy uzlů připravovat (Büttner a kol., 2004).

Následuje ukázka návštěvníka uzlu `Return`, který reprezentuje klíčové slovo *return*. Tento návštěvník na uzlu definuje metodu `Run`, která slouží ke spuštění interpretace uzlu. Za pozornost stojí klíčové slovo `dynamic`, které slouží k implementaci *dynamic dispatcheru*. Návštěvníci uzlů AST definují uzlům metodu `Run` nebo metodu `GetValue`.

```
public static void Run(Return node, Scope s) {
   s.ReturnValue = Visitor.GetValue((dynamic)node.Valuable, s);
   s.ReturnOccured = true;
}
```



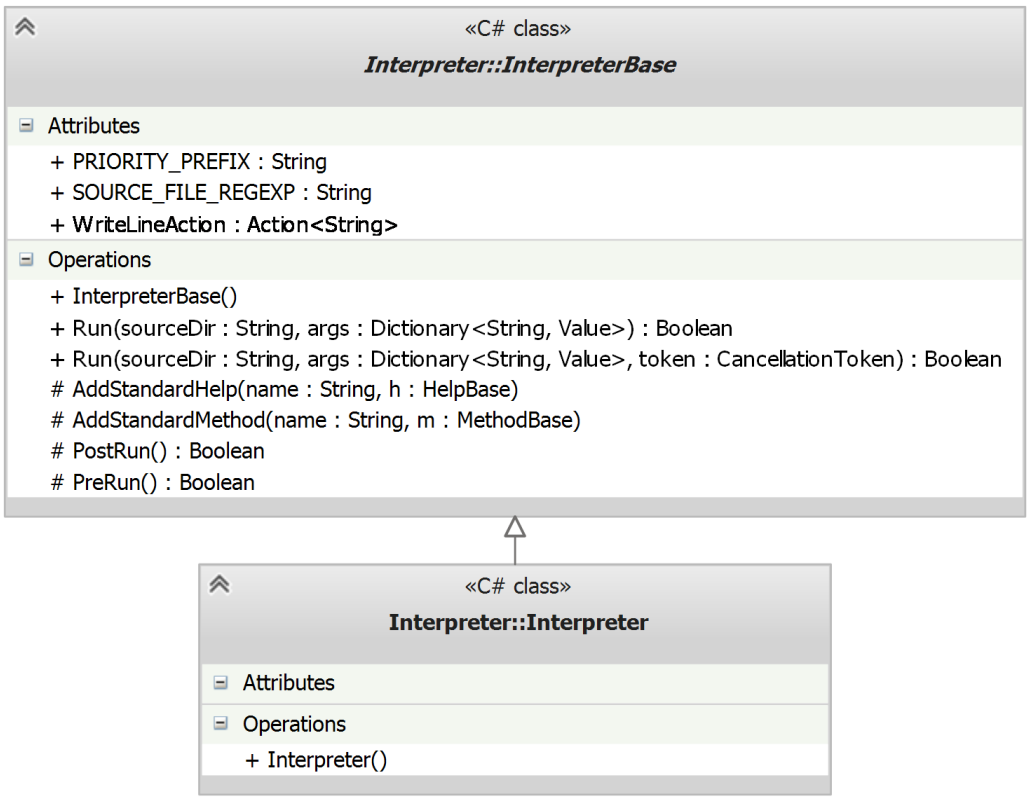
Obrázek 7.7: Diagram struktury návrhového vzoru *Visitor Pattern*
Obrázek použit z dostupného zdroje pod licencí CC BY-SA 3.0 `https://upload.wikimedia.org/wikipedia/en/e/eb/Visitor_design_pattern.svg`.

Standardní metody

Abstraktní třída `InterpreterBase` definuje základ jazyka. Její přetížená metoda `Run` umožňuje překlad a interpretaci zdrojového kódu programu. Zdrojovým kódem zde není jeden soubor, ale celá složka, jejíž cesta se použije jako parametr metody `Run`. V této složce a jejích podsložkách se najdou všechny soubory, které vyhovují `SOURCE_FILE_REGEXP` , a ty se použijí jako zdrojové soubory jazyka. Dalším parametrem metody `Run` je kolekce vstupních argumentů programu. Posledním volitelným parametrem je `CancellationToken`, který je využit k přerušení asynchronního zavolání interpretu.

Tato třída definuje protected metody `AddStandardMethod` a `AddStandardHelp`, pomocí níž mohou její potomci nadefinovat standardní metody jazyka a stránky nápovědy. Součástí definice každé standardní metody je také nápověda k ní. Nápovědy, které nejsou svázané k jediné metodě je možné definovat právě pomocí `AddStandardHelp`. Metod `AddStandardMethod` a `AddStandardHelp` využívá třída `Interpreter`, která např. dodefinovává standardní metody jazyka pro

práci s poli, soubory, výstupem a čísly (viz Obrázek 7.8).

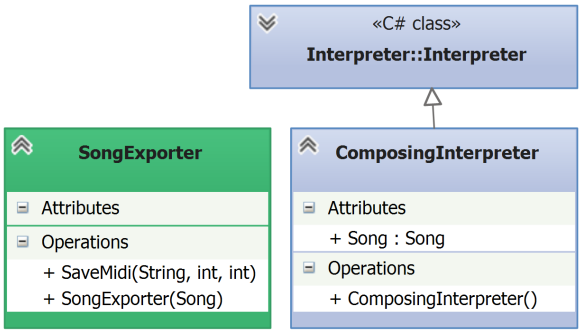


Obrázek 7.8: Detail public (+) a protected (#) členů tříd `Interpreter` a `InterpreterBase`

7.3 ComposingInterpreter

Třída `ComposingInterpreter` rozšiřuje `Interpreter` o práci s hudbou. Dodefinovává hudební standardní metody např. pro práci s melodií, pedály a rytmem, rozšiřuje třídu `Interpreter` o stránky nápovědy a definuje getter `Song`, pomocí kterého je možné po dokončení interpretace získat vytvořenou skladbu, která je reprezentovaná třídou `Song`.

S vytvořenou skladbou také souvisí její export. Funkcionalita exportu je však definovaná v oddělené třídě `SongExporter`, která umožňuje exportování skladby do formátu MIDI. `SongExporter` k tomu definuje metodu `SaveMidi` (viz Obrázek 7.9).



Obrázek 7.9: Přehled hlavních tříd projektu `ComposingInterpreter`

7.3.1 Song

Třída `Song` představuje abstrakci nad formátem MIDI. Uložení skladby ve formátu `.song` je naprogramováno pomocí *XML serializace* třídy `Song`.

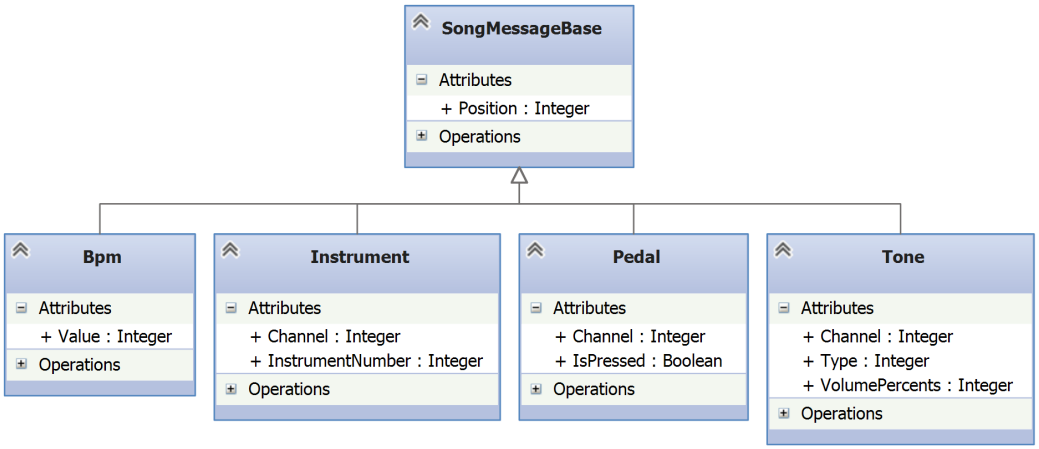
Zprávy

Data skladby jsou ve třídě reprezentována pomocí tzv. zpráv, což jsou potomci abstraktní třídy `SongMessageBase` (viz Obrázek 7.10). Tyto zprávy jsou abstrakcí nad MIDI zprávami a díky abstrakci je možné v dalších verzích programu nahradit MIDI jiným formátem. Mezi zprávy patří například zahrání tónu, stisknutí či puštění sustain pedálu či změna hudebního nástroje. Třída `Song` definuje metodu `Add`, pomocí které je možné do skladby zprávy přidávat.

Zprávy jsou uloženy v kolekci typu `SongMessageList`, což je potomek .NET typu `SortedList`, který rozšiřuje o schopnost *XML serializace*. Každá zpráva nese informaci o své pozici v skladbě pomocí `SongMessageBase` členu `Position` a zprávy jsou do kolekce `SongMessageList` přidávány tak, aby byly časově seřazené. Tento přístup je zvolen z důvodu, aby bylo v dalších verzích programu možné, skladbu přehrávat (streamovat), zatímco se vytváří.

7.3.2 Export do formátu MIDI

Export třídy `Song` je delegován na třídu `SongExporter`. Tato třída k exportu do MIDI využívá třídu `MessageProcessor`, která realizuje *návštěvníka* zpráv.



Obrázek 7.10: Třída `SongMessageBase` a její potomci

Tyto potomky třídy `SongMessageBase` návštěvník rozšiřuje o metodu `Process`, která ze zprávy vytvoří MIDI zprávu, kterou vloží do sekvence ve formátu MIDI.

K realizaci návštěvníka je využit *dynamic dispatcher*. Tím, že je práce delegována na návštěvníka `MessageProcessor`, může být převod do formátu MIDI velmi čistě rozdělen do několika metod, které reprezentují návštěvníky. `SongExporter` pak jen navštíví všechny zprávy a vytvoří MIDI sekvenci:

```
/* inicializace návštěvníka */
MessageProcessor processor =
   new MessageProcessor(Song, track, trans, speed, MARGIN_LEFT);

/* navštívení všech zpráv skladby pomocí dynamic dispatcher */
foreach (SongMessageBase e in Song.Messages) {
    processor.Process((dynamic) e);
}
```

7.3.3 Použití

Následuje ukázka použití projektu ComposingInterpreter, ať už v projektu SongMaker nebo ve formě DLL knihovny. Následující kód interpretuje jazyk *Composing*, uloží skladbu do souboru `file.song` a exportuje do souboru `file.midi`:

```
ComposingInterpreter interpreter = new ComposingInterpreter();

/* Spusť interpretaci */
if (interpreter.Run("source-dir/", args)) {

    Console.WriteLine("Interpretation succesful");

    /* Ulož složenou skladbu do souboru */
    interpreter.Song.SaveToFile("file.song");

    /* Exportuj se 100% rychlostí a s transpozicí o 3 půltóny */
```

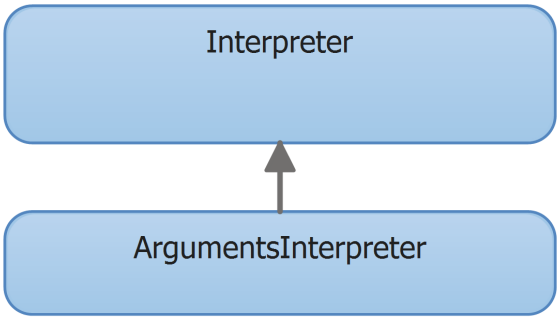
```
    new SongExporter(interpreter.Song).SaveMidi("file.midi", 100, 3);

} else {
    Console.WriteLine("Interpretation failed");
}
```

7.4 SongMaker

Projekt `SongMaker` slouží jako GUI, které zpřístupňuje funkce `ComposingInterpreter`.

7.4.1 ArgumentsInterpreter

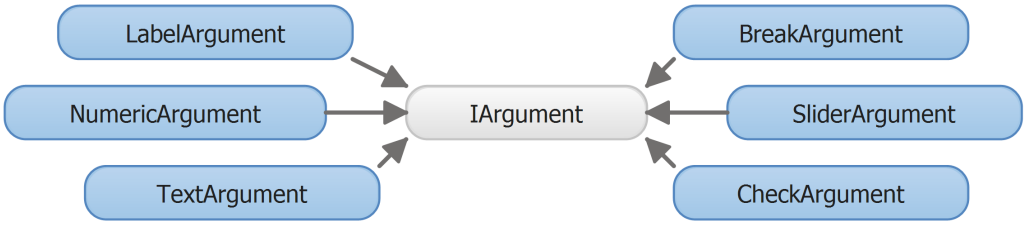
Předtím, než popíšeme návrhový vzor uživatelského rozhraní, zmíníme třídu `ArgumentsInterpreter`. Tato třída je potomkem `Interpreter` (viz Obrázek 7.11) a podobně jako `ComposingInterpreter` dodefinovává svému předkovi standardní metody. Tentokrát ne pro vytvoření skladby, ale pro vytvoření ovládacích prvků k přizpůsobení šablon.



Obrázek 7.11: Závislost tříd `ArgumentsInterpreter` a `Interpreter`

Ovládací prvky, kterým říkáme **argumenty**, jsou reprezentovány třídami, které implementují rozhraní `IArgument` (viz Obrázek 7.12). Pomocí třídy `ArgumentsInterpreter` lze kolekci argumentů pro danou šablonu snadno získat.

Tyto argumenty reprezentující ovládací prvky jsou před kompozicí skladby převedeny na hodnotové argumenty interpretu. Někteří potomci `IArgument` jsou při převodu na argumenty interpretu ignorovány a slouží jen jako grafický prvek v GUI (například popisek *LabelArgument* nebo znak nového řádku *BreakArgument*).

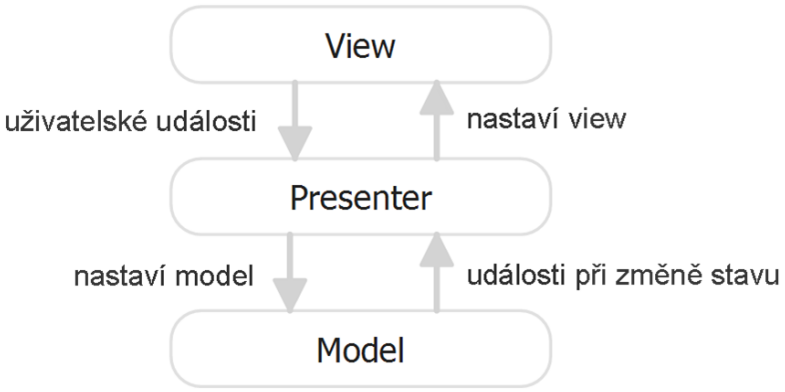


Obrázek 7.12: Potomci rozhraní `IArgument`
Každý z těchto potomků reprezentuje ovládací prvek použitý k přizpůsobení kompozice.

7.4.2 Model-view-presenter

Model-view-presenter (MVP) je návrhový vzor odvozený od vzoru model-view-controller (MVC), který je využit k vytváření GUI.

Cílem tohoto návrhového vzoru je striktnější oddělení view a presenteru. Presenter představuje komponentu mezi modelem a view (viz Obrázek 7.13).



Obrázek 7.13: Tok dat v návrhovém vzoru Model-view-presenter
Veškerá komunikace mezi view a modelem probíhá prostřednictvím presenteru.

MVP je návrhový vzor, který pomáhá automatizovat testování a zlepšuje oddělení zodpovědnosti rozdělením na následující komponenty:

- **Model** představuje data, která se mají zobrazit v GUI. Tvoří jakýsi backend aplikace.
- **View** definuje rozhraní, které zobrazuje data uživateli a přeposílá příkazy uživatele do presenteru. Rozhraní, které view definuje, je nezávislé na technologii, pomocí které je view implementováno. Toto rozhraní představuje data, která jsou ve view zobrazena.
- **Presenter** je prostředník mezi view a modelem. Získává data z modelu a deleguje je do view. Zpracovává také uživatelský vstup z modelu a volá funkce modelu. Pro komunikaci s view využívá rozhraní, které view implementuje a které je nezávislé na použité technologii view. Presenter je pro view přístupný pomocí rozhraní.

Použití MVP má oproti tradičním návrhovým vzorům (jako je např. MVC, které je porovnáno s MVP v Sekci 7.4.2) několik výhod (Zhang a Luo, 2010):

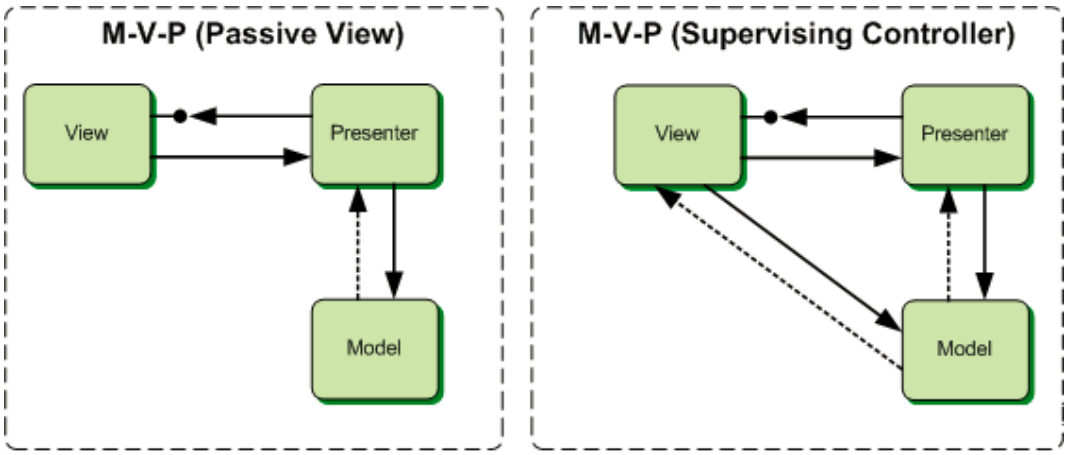
1. View nepřistupuje přímo k modelu. Díky tomuto faktu na sobě view a model nejsou závislé. To znamená, že pokud by se jeden z nich změnil, druhý se měnit nemusí. Tím je architektura aplikace více flexibilní.
2. Presenter ignoruje technologickou implementaci view. Z toho důvodu je pak možné snadno nahradit UI technologii za jinou, jako například místo Windows Forms použít WPF nebo Web Forms, aniž by bylo třeba měnit další části aplikace. Díky tomu také může mít aplikace snadno více uživatelských rozhraní.
3. View je velmi vhodně navržené pro testovací účely. Při tradičních přístupech je nemožné testovat view nebo business logiku před tím, než je druhá komponenta dokončena, kvůli jejich silné vzájemné závislosti. Všechny tyto problémy jsou pomocí MVP vyřešené. V tomto vzoru mezi view a modelem závislost není. Vývojář pak díky tomu může testovat view i model odděleně.

Passive view a supervising controller

Existují dva základní přístupy, kterými se dá MVP realizovat.

Při **passive view** je kladeno více práce na presenter, který odděleně přistupuje k datům modelu a ta nastavuje jako data view. Mezi modelem a view tak neexistuje žádná interakce a view nemá znalost o typech, které jsou využívány v modelu.

Při **supervising controller** view využívá hodnotové typy modelu a docílí tak jednoduchého provázání dat. Presenter tak může aktualizovat data modelu, přistoupit k objektu, který tato data reprezentuje, a ten přímo delegovat do view. Výjimkou pak je, když je logika uživatelského nastavení příliš komplexní a presenter tak například musí ještě dynamicky nastavit barvu ovládacích prvků nebo změnit jejich viditelnost (viz Obrázek 7.14).



Obrázek 7.14: Rozdíly mezi passive view a supervising controller
V passive view je interakce s modelem řízená jen pomocí presenteru. View je aktualizováno presenterem pomocí rozhraní. V supervising controller view k vykreslování využívá hodnotové typy modelu. View je updatováno pomocí presenteru s využitím těchto typů. Zdroj: MSDN:MVC

Srovnání s MVC

Oba návrhové vzory MVP a MVC se snaží oddělit práci různých aplikačních komponent. Mají však několik rozdílů (viz Obrázek 7.15):

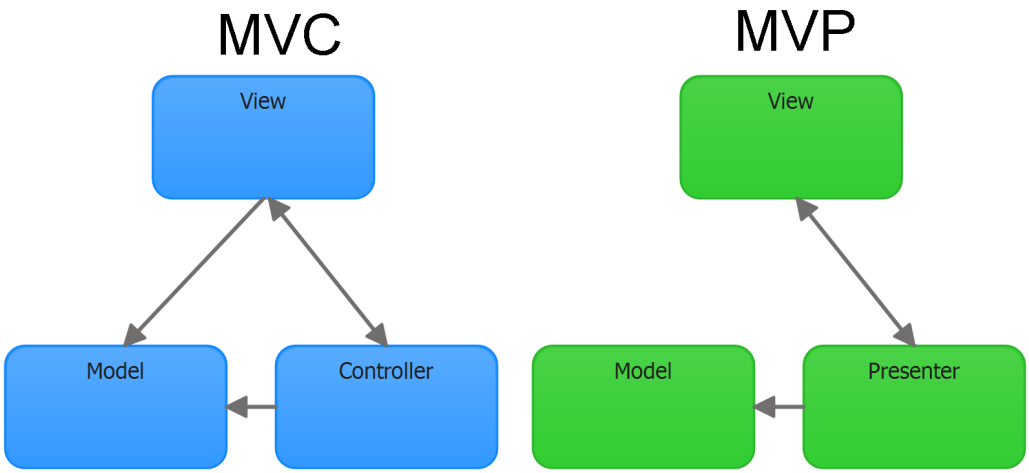
Hlavním rozdílem mezi těmito vzory je, že u MVC bývá velmi úzce provázáno view a model. To někdy do takové míry, že když se změní jedno z dvojice view nebo model, provede odpovídající změnu i druhé z dvojice, aniž by tuto práci provedl controller (Jain a kol., 2015). Tím pádem je view navrženo tak, aby představovalo vizualizaci odpovídajícího objektu modelu.

Při použití MVP presenter přistupuje k view pomocí rozhraní, které view implementuje. Toto rozhraní není závislé na technologii, kterou je view implementováno, ale představuje logický pohled na view. U MVC controller přistupuje přímo k uživatelským prvkům view a je tedy závislý na implementaci view.

V MVP je view pasivní. Jeho interakce s modelem nejdříve musí projít přes presenter. View v MVC je aktivní, protože dokáže poslat požadavek o přístup

datům přímo modelu a při modifikaci modelu se samo dokáže modifikovat bez použití presenteru.

Testovat MVP je snadnější než testovat MVC (Qureshi a Sabir, 2014).

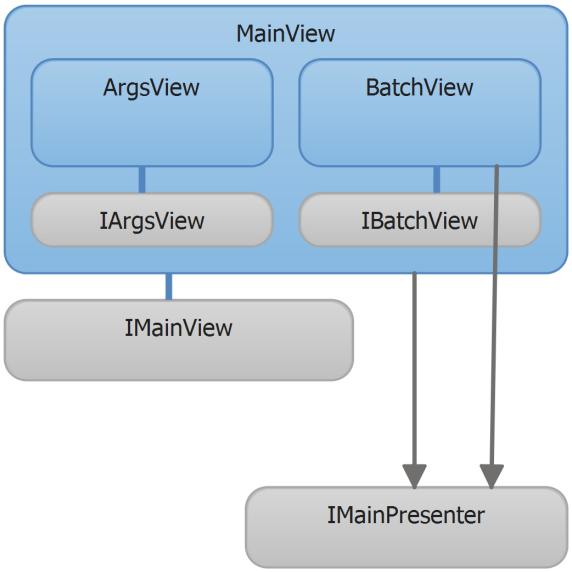


Obrázek 7.15: Srovnání závislostí komponent vzorů model-view-controller a model-view-presenter

Obrázek je vytvořen podle Qureshi a Sabir (2014).

7.4.3 Implementace MVP

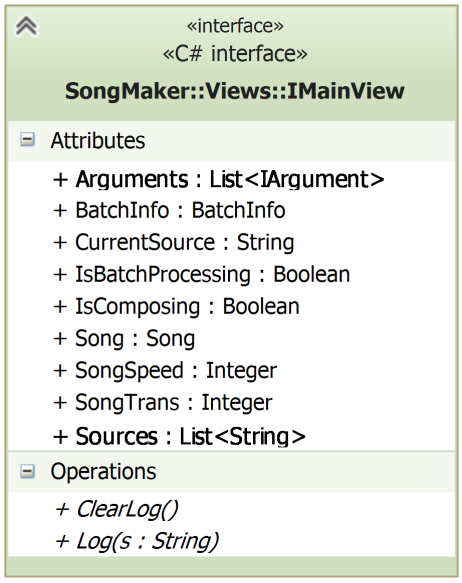
View



Obrázek 7.16: Závislosti tříd a rozhraní představující view

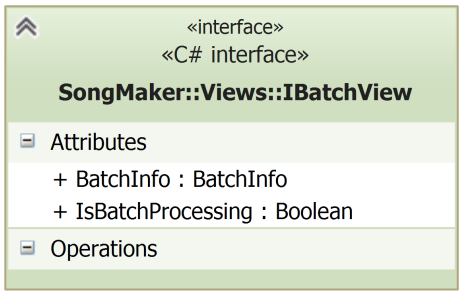
View je reprezentováno pomocí *Windows Forms*. Třída `MainView` je potomkem třídy `Form` a vytváří tak hlavní okénko uživatelského rozhraní. Dále implementuje rozhraní `IMainView` (viz Obrázek 7.16) představující logiku dat, která

jsou v `MainView` zobrazena (viz Obrázek 7.17). Díky tomuto rozhraní je view přístupné presenteru. Když uživatel klikne na tlačítko UI nebo provede jinou akci, `MainView` zavolá příslušnou metodu, kterou definuje presenter pomocí rozhraní IMainPresenter (viz Obrázek 7.21).



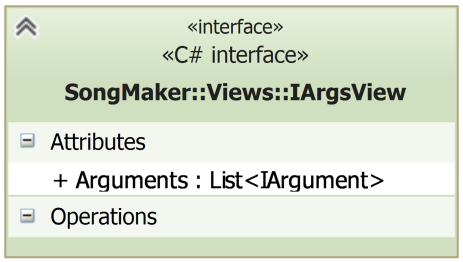
Obrázek 7.17: Atributy a operace rozhraní `IMainView`

Třída `BatchView` je také potomkem `Form` a představuje okénko dávkového exportu. `BatchView` je na `MainView` nezávislé, což pomáhá v přehlednosti kódu. `MainView` k ní přistupuje pomocí jejího rozhraní `IBatchView` (viz Obrázek 7.18). Díky tomuto rozhraní může `MainView` na třídu `BatchView` delegovat funkcionalitu atributů `BatchInfo` a `IsBatchProcessing`. Třída `BatchView` pro spuštění dávkového exportu využívá rozhraní presenteru.

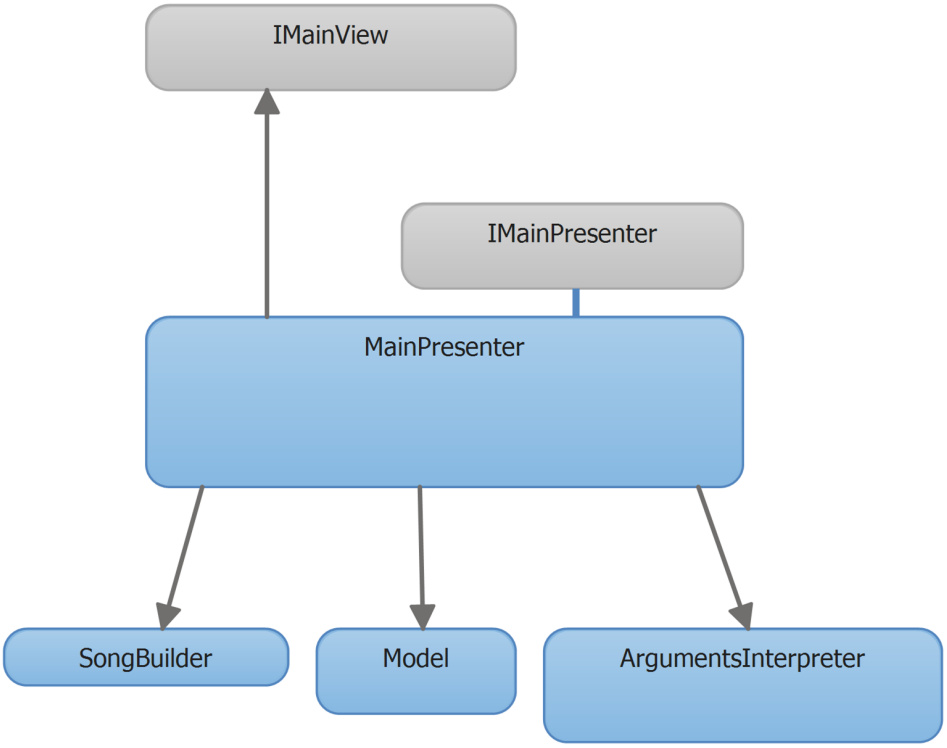


Obrázek 7.18: Atributy rozhraní `IBatchView`

Panel s ovládacími prvky na nastavení šablony je reprezentován třídou `ArgsView`, která je potomkem `UserControl`. Třída je, podobně jako `BatchView`, nezávislá na `MainView`, kterému zpřístupňuje svoji funkcionalitu pomocí rozhraní `IArgsView` (viz Obrázek 7.19). Funkcionalita atributu `Arguments` třídy `MainView` je delegována skrz rozhraní na `ArgsView`.



Obrázek 7.19: Atributy rozhraní `IArgsView`



Obrázek 7.20: Závislosti presenteru reprezentovaného třídou `MainPresenter`

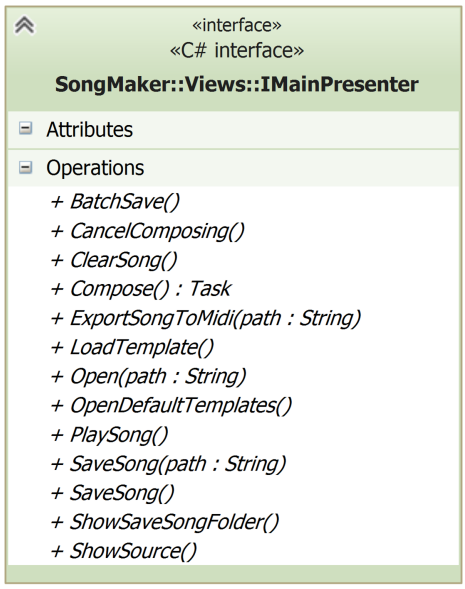
Presenter

Presenter je reprezentován třídou `MainPresenter`. Tato třída implementuje rozhraní `IMainPresenter`, pomocí kterého ho může využívat view (viz Obrázek 7.21). Implementace rozhraní dosáhne za použití modelu, který je reprezentován třídami `SongBuilder`, `Model` a `ArgumentsInterpreter` (viz Obrázek 7.20).

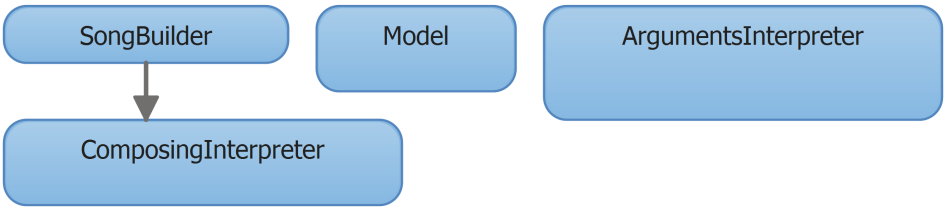
Model

Model je představen třemi třídami, které tím tvoří logickou část aplikace (viz Obrázek 7.22).

Třída **`SongBuilder`** obaluje funkcionalitu a představuje tzv. **wrapper** pro `ComposingInterpreter`. Definuje tyto metody:



Obrázek 7.21: Atributy rozhraní `IMainPresenter`



Obrázek 7.22: Třídy tvořící model a jejich závislosti

```
public Song Build(CancellationToken token);
public Task<Song> BuildAsync(CancellationToken token);
```

`SongBuilder` kromě synchronního vytvoření skladby umožňuje skladbu vytvořit asynchronně pomocí metody `BuildAsync`. V takovém případě využije *Futures pattern*[1], který je implementován .NET Frameworkem. Místo synchronního vrácení objektu třídy `Song` pak vrací objekt, který objekt třídy `Song` dokáže poskytnout v budoucnu, tzv. „future“. Díky využití této technologie může probíhat interpretace, zatímco uživatel využívá dále grafické rozhraní, aniž by bylo třeba využít procesu throttling (Loidl, 2012).

Při vytváření skladby používáme parametr typu **`CancellationToken`**, díky kterému je možné proces komponování v jakémkoliv okamžiku přerušit.

Díky třídě `SongBuilder` by byl také program snadněji upravitelný, pokud bychom se rozhodli komponovat hudbu jiným způsobem. V takovém případě by pak stačilo změnit **wrapper** a zbytek projektu *SongMaker* by zůstal beze změny.

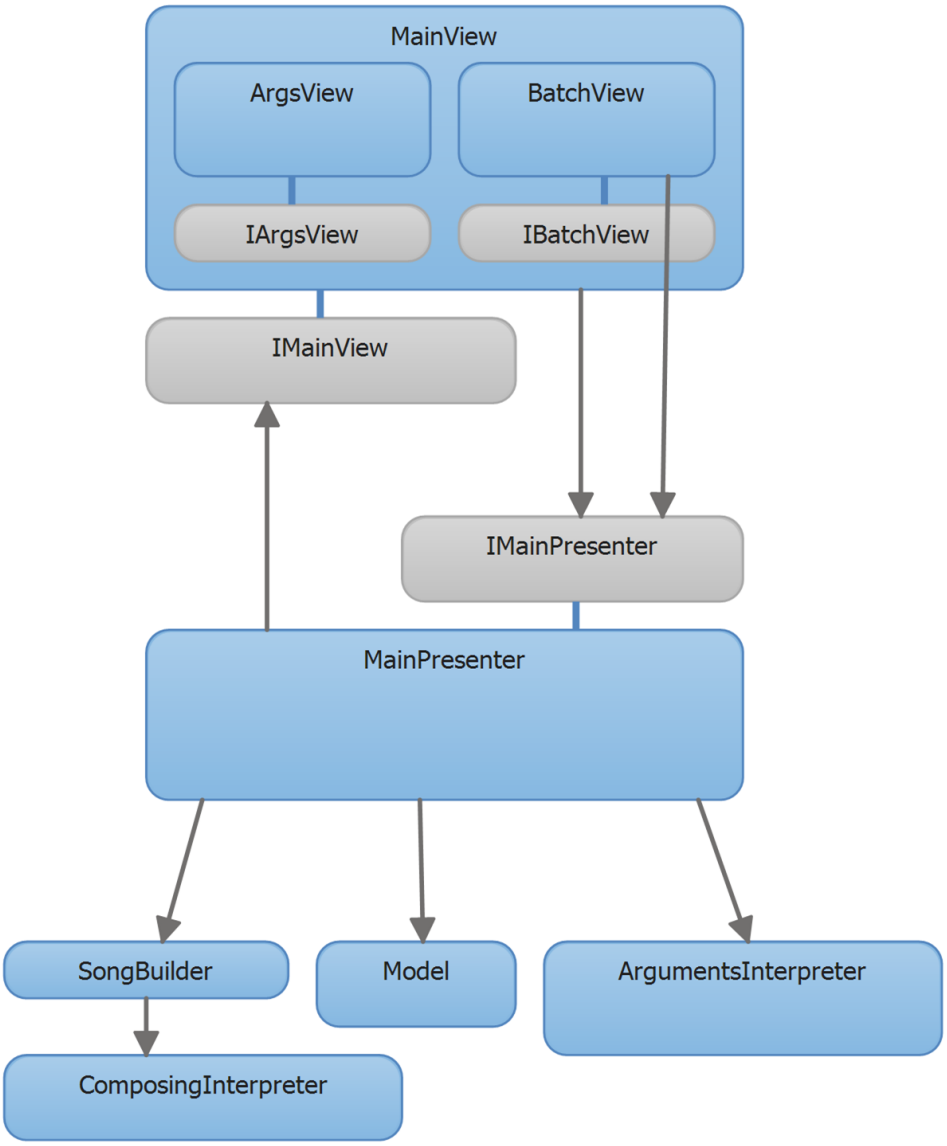
Třída **`ArgumentsInterpreter`** pomáhá s vytvořením uživatelských prvků.

Zbylá funkcionalita modelu, kterou využívá presenter, je ve třídě `Model`.

[1] `https://msdn.microsoft.com/en-us/library/ff963556.aspx`

Shrnutí

Separací uživatelského rozhraní od logiky celé aplikace jsme dosáhli kódu, který je možné snadněji testovat a modifikovat. Všechny tři komponenty model, view a presenter jsou velmi čistě provázány (viz Obrázek 7.23)



Obrázek 7.23: Přehled architektury projektu `SongMaker` navržené pomocí vzoru MVP

Závěr

Cílem práce bylo vytvoření aplikace, která by umožňovala hudební kompozici pomocí programovacího jazyka a zároveň by byla přístupná uživatelům bez hudebního či technického zázemí.

Vytvořili jsme hudební programovací jazyk *Composing*, který jsme ke skládání hudby navrhli. Jazyk má schopnost definovat jednu metodu vícekrát, kdy se při jejím zavolání použije její náhodná definice. Tímto přístupem můžeme jednoduše pomocí jazyka dosáhnout stejného výsledku, jako při popsání hudby pomocí formálních gramatik, kterých se k algoritmickému skládání hudby často používá. Programovací jazyk navíc umožňuje skladateli využívat množství standardních metod na práci s hudbou, čímž má skladatel velkou volnost pro své kompozice.

V porovnání s některými jinými programovacími jazyky ke skládání hudby je náš jazyk podobný syntaxí C a je navržen tak, aby mohl být používán i skladateli bez programátorských zkušeností.

Pomocí standardních metod pro práci s hudbou nadefinovaných v jazyce je snadné algoritmicky skládat hudbu, která „dobře zní“. Díky faktu, že v interpretaci jazyka je využit velký prvek náhodnosti, může jeden program při svých spuštěních vyprodukovat velké množství různých skladeb.

Dalším cílem bylo zpřístupnit aplikaci pro uživatele bez technického a hudebního zázemí. Toho jsme dosáhli grafickým uživatelským rozhraním *SongMaker*, pomocí kterého lze načítat programy vytvořené skladateli v jazyce, ty pomocí grafických elementů přizpůsobovat a využít je k produkování hudby. Vyprodukovanou hudbu je možné uložit a znovu ji otevřít z rozhraní a je možné ji přímo z rozhraní exportovat do formátu MIDI a přehrát.

V grafickém rozhraní jsou již předdefinované šablony pro kompozici hudebních žánrů jako ragtime či waltz. Součástí rozhraní je také šablona pro zahrání doprovodu ke skladbě pomocí nadefinování jejích akordů, čehož lze využít jako podkladu ke zpěvu.

Rozhraní má podporu také pro dávkový export, pomocí kterého je možné vytvořit jedním kliknutím velmi dlouhé playlisty.

Aplikace může být využita hudebníky k usnadnění a urychlení procesu kompozice skladby či k produkci hudebních materiálů. Také může být užitečná pro vývojáře počítačových her či pro filmové tvůrce, kteří si mohou bez hudebních dovedností zkomponovat unikátní skladby a využít je jako hudbu na pozadí. Aplikace ale stejně může být využita k osobním účelům - k vygenerování přizpůsobených playlistů pro poslech hudby či jako hudební nástroj k doprovodu ke zpěvu.

V budoucích verzích aplikace je možnost přidat funkcionalitu komponování hudby v reálném čase. Bylo by pak možné produkovat hudbu a ve stejný čas ji komponovat. To by přineslo několik výhod. Skladby by nemusely být omezené délkou a bylo by tak možné hrát hudbu neustále. Navíc by pak bylo možné do procesu kompozice skladby vstupovat a za běhu například měnit akordy či přepínat motivy. Toho by šlo docílit i pomocí externích MIDI zařízení, jako je např. elektrické piano, která by tak mohla do procesu kompozice velmi dobře vstupovat.

Dále je možné přidávat další užitečné standardní metody pro práci s hudbou, které proces komponování ještě zjednoduší a schopnost *live coding*.

Možným vylepšením v budoucích verzích je také rozšiřování repertoáru napro-

gramovaných šablon pro komponování různých hudebních žánrů. Bylo by vhodné umožnit skladatelům tyto šablony mezi sebou sdílet a umožnit jejich stažení z cloudu.

Pro komponování hudby pomocí vytvořených šablon by bylo vhodné vytvořit i internetové uživatelské rozhraní umožňující kompozici skladeb přímo v prohlížeči bez nutnosti instalace programu.

Programování hudby je obor, který si určitě zaslouží více pozornosti. Jazyk *Composing* a aplikace *Song Maker* přispívá do tohoto oboru jako jedno z rychlých a efektivních řešení a věříme, že podnítí k dalšímu výzkumu.

Seznam použité literatury

Beaumont, D. a Stepney, S. (2009). Grammatical evolution of l-systems. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 2446–2453. IEEE.

Blackwell, A. a Collins, N. (2005). The programming language as a musical instrument. *Proceedings of PPIG05 (Psychology of Programming Interest Group)*, **3**, 284–289.

Brandon Bray, N. B. (2012). Announcing the release of .net framework 4.5 rtm. `https://blogs.msdn.microsoft.com/dotnet/2012/08/15/announcing-the-release-of-net-framework-4-5-rtm-product-and-source-code/`. (Navštíveno 7.5.2017).

Büttner, F., Radfelder, O., Lindow, A. a Gogolla, M. (2004). Digging into the visitor pattern. In *SEKE*, pages 135–141.

Buxton, W., Reeves, W., Baeker, R. a Mezei, L. (1978). The use of hierarchy and instance in a data structure for computer music. *Computer Music Journal*, **2**(2), 10–20.

Edwards, M. (2011). Algorithmic composition: computational thinking in music. *Communications of the ACM*, **54**(7), 58–67.

Eldridge, A. C. (2005). Extra-music (ologic) al models for algorithmic composition. In *Workshops on Applications of Evolutionary Computation*, pages 557–562. Springer.

Jain, N., Mangal, P. a Mehta, D. (2015). Angularjs: A modern mvc framework in javascript. *Journal of Global Research in Computer Science*, **5**(12), 17–23.

Jeanmart, S., Gueheneuc, Y.-G., Sahraoui, H. a Habra, N. (2009). Impact of the visitor pattern on program comprehension and maintenance. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 69–78. IEEE Computer Society.

Langston, P. (1989). Six techniques for algorithmic music composition. In *15th International Computer Music Conference (ICMC), Columbus, Ohio, November*, pages 2–5. Citeseer.

Loidl, H.-W. (2012). Parallel programming in c#. *Heriot-Watt University, Edinburgh*.

Maeda, K. (2012). Performance evaluation of object serialization libraries in xml, json and binary formats. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on*, pages 177–182. IEEE.

Marguerie, F., Eichert, S. a Wooley, J. (2008). *LINQ in Action*. Manning.

McCartney, J. (2002). Rethinking the computer music language: Supercollider. *Computer Music Journal*, **26**(4), 61–68.

McCormack, J. (1996). Grammar based music composition. *Complex systems*, **96**, 321–336.

MSDN:MVC. Model-view-presenter. `https://msdn.microsoft.com/en-us/library/ff709839.aspx`. (Navštíveno 7.5.2017).

MSDN:.NET Framework Versions and Dependencies. .net framework versions and dependencies. `https://msdn.microsoft.com/en-us/library/bb822049(v=vs.110).aspx`. (Navštíveno 7.5.2017).

Qureshi, M. a Sabir, F. (2014). A comparison of model view controller and model view presenter. *arXiv preprint arXiv:1408.5786*.

Roads, C. (1979). Grammars as representations for music. *Computer Music Journal*, **3**(1), 45–55.

Syromiatnikov, A. a Weyns, D. (2014). A journey through the land of model-view-design patterns. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 21–30. IEEE.

Tokui, N., Iba, H. a kol. (2000). Music composition with interactive evolutionary computation. In *Proceedings of the 3rd international conference on generative art*, volume 17, pages 215–226.

Wang, G. (2008). *The ChucK audio programming language."A strongly-timed and on-the-fly environ/mentality"*. Princeton University.

Zhang, Y. a Luo, Y. (2010). An architecture and implement model for model-view-presenter pattern. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 8, pages 532–536. IEEE.

Seznam obrázků

Seznam použitých zkratek

- **MIDI** Musical Instrument Digital Interface
- **GUI** Graphic User Interface
- **GM** General MIDI
- **DLL** Dynamic-link library
- **SMF** Standard MIDI File
- **AST** Abstract Syntax Tree
- **MVP** Model-View-Presenter
- **MVC** Model-View-Controller
- **MVVM** Model-View-ViewModel

Přílohy

Obsah přiloženého CD:

source	Zdrojové kódy aplikace
setup	Instalační balíček aplikace
documentation	Uživatelská dokumentace
samples	Ukázky zkomponovaných skladeb